Mapping Unstructured Grid Computations to Massively Parallel Computers

Steven Warren Hammond

This work has been supported in part by an IBM doctoral fellowship, by DARPA via Cooperative Agreement NCC 2-387 between NASA, and by the University Space Research Association (USRA).

MAPPING UNSTRUCTURED GRID COMPUTATIONS TO MASSIVELY PARALLEL COMPUTERS

By

Steven Warren Hammond

Thesis Submitted to the Graduate Faculty

of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Approved by the
Examining Committee:

Robert S. Schreiber, Co-Chairman

Joseph E. Flaherty, Co-Chairman

W. Randolph Franklin, Member

Mark S. Shephard, Member

Boleslaw Szymanski, Member

Rensselaer Polytechnic Institute
Troy, New York

February 1992
(For Graduation August 1992)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

To Lisa

ACKNOWLEDGEMENT

First of all I want to thank my advisor, Rob Schreiber, for his support and encouragement during the last three years. His enthusiasm, knowledge, and high standards are an inspiration. Even when he had more than enough of his own work to do, he always made time to work with me on my thesis. Rob, thanks for your patience, guidance, and above all, for being a friend.

I would also like to thank Joe Flaherty, W. Randolph Flanklin, Mark Shephard, and Bolek Szymanski for serving on my committee.

I am deeply indebted to Tim Barth. Our collaboration on the parallel unstructured Euler code provided much of the motivation for this research. Also, our food logs, pints at the Tied House, Giants' games, and BBQ's provided necessary diversions.

I want to thank Marsha Berger, Rupak Biswas, and Noel Nachtigal for reading drafts of this thesis. They suggested many improvements and clarifications that contributed greatly to this thesis. Additionally, Rupak printed the final version of this and turned in while I was in France.

I want to thank Kyra Lowther for providing ample amounts of "hand-holding" as I learned the Symbolics, lisp, *lisp, and the CM-2 all at once. Also, while our programs ran, we swapped a recipe or two. Denny Dahl provided me with alpha releases of the communication compiler software and invaluable insight into the way messages are delivered on the CM-2.

I thank my cycling friends Eugene Cordero, Roland Freund, Roger Strawn, and Leigh Ann Tanner for giving me an opportunity to work out frustrations on two wheels.

This work has been supported in part by an IBM doctoral fellowship, by

DARPA via Cooperative Agreement NCC 2-387 between NASA, and by the University Space Research Association (USRA). Finally, I thank the NAS project at NASA Ames Research Center for providing me with (much more than my share of) time and disk space on the CM-2.

ABSTRACT

This thesis investigates the mapping problem: assign the tasks of a parallel program to the processors of a parallel computer such that the execution time is minimized.

First, a taxonomy of objective functions and heuristics used to solve the mapping problem is presented. Next, we develop a highly parallel heuristic mapping algorithm, called *Cyclic Pairwise Exchange* (CPE), and discuss its place in the taxonomy. CPE uses local pairwise exchanges of processor assignments to iteratively improve an initial mapping. A variety of initial mapping schemes are tested and *recursive spectral bipartitioning* (RSB) followed by CPE is shown to result in the best mappings. For the test cases studied here, problems arising in computational fluid dynamics and structural mechanics on unstructured triangular and tetrahedral meshes, RSB and CPE outperform methods based on simulated annealing. Much less time is required to do the mapping and the results obtained are better. Compared with random and naive mappings, RSB and CPE reduce the communication time twofold for the test problems used.

Finally, we use CPE in two applications on a CM-2. The first application is a data parallel mesh-vertex upwind finite volume scheme for solving the Euler equations on 2-D triangular unstructured meshes. CPE is used to map grid points to processors. The performance of this code is compared with a similar code on a Cray-YMP and an Intel iPSC/860. The second application is parallel sparse matrix-vector multiplication used in the iterative solution of large sparse linear systems of equations. We map rows of the matrix to processors and use an inner-product based matrix-vector multiplication. We demonstrate that this method is an order of magnitude faster than methods based on scan operations for our test cases.

CHAPTER 1
INTRODUCTION

In massively parallel computer systems, tens of thousands of processors are interconnected into single units. These systems promise high peak performance. For example, the Connection Machine [37] CM-2 has 64K bit-serial processors and a peak rate of 13.7 Gflops (1.71 Gflops on 8K processors) in 64-bit arithmetic. However, an imprudent assignment of tasks to processors can cause unnecessary interprocessor communication. The communication time – the amount of time spent moving data between processors – can dominate the computation time and thus limit the realized performance.

Many projects have demonstrated that massively parallel architectures are effective at solving discretized Partial Differential Equations (PDEs) when the discretizing grids are fixed and topologically simple, Cartesian for example [2, 9, 15, 24, 40, 49, 51, 53, 54, 77]. Parallel numerical techniques used to solve PDEs decompose the computation into concurrent *tasks* that are associated with the grid points of the discretizing grid, one task per grid point. A *task* is a set of local data, computations, and communications (input and output data). The tasks repeatedly perform computations on local data and then exchange locally stored or computed data with other tasks. These steps are repeated until some desired solution is achieved – perhaps thousands or tens of thousands of times (for large computations). The tasks associated with Cartesian grids can be optimally mapped to parallel computers whose interconnection is a grid or torus since the topologies are identical. Multi-dimensional gray codes can be used when mapping tasks associated with Cartesian grids to hypercubes.

Here we study the use of parallel architectures for more difficult problems, where the discretizing grid is arbitrary, but static. This thesis investigates mapping the tasks associated with the solution of unstructured grid problems to the processors of a parallel computer such that the execution time is minimized. Optimal graph mapping is NP-complete [13]. We can, however, use heuristics.

1.1 Thesis Outline

This thesis is organized as follows. In the remainder of the Introduction, we define terminology and notation from graph theory that is used throughout the thesis. Next, we discuss the architecture and operation of the Connection Machine CM-2. It is used to experimentally validate CPE. Finally, we present the contributions of this thesis.

The remaining chapters are as follows. In Chapter 2 we give a formal definition of the mapping problem. Also, we present a taxonomy of objective functions and heuristic algorithms used to solve the mapping problem and discuss related and prior work.

In Chapter 3 we develop the CPE heuristic, a highly parallel iterative mapping algorithm. CPE starts with some initial assignment of tasks to processors. To improve an initial mapping, CPE uses an iterative parallel pairwise exchange algorithm in which pairs of neighboring processors may exchange the tasks mapped to them. Several different initial mappings are tested. We describe the implementation of CPE on the CM-2 and verify its capabilities with a variety of test cases. For very large, very irregular problems arising in 2-D flow around complex multi-component airfoils and 3-D flow around aircraft, it has achieved excellent results. The method has outperformed parallel methods based on simulated annealing (SA). Compared to SA, CPE requires up to a factor of six less time to do the mapping and the results obtained are better. By this we mean that, for our test cases, an application requires less execution time when using a mapping produced by CPE than when using a mapping produced by SA. Compared with random and naive mappings, CPE reduces the execution time twofold for realistic, large, highly irregular, and stretched meshes.

In Chapters 4 and 5 we incorporate CPE into two applications. In Chapter 4 we develop a data parallel implementation of Barth and Jespersen's mesh-vertex upwind finite volume scheme for solving the Euler equations on triangular unstructured meshes [5]. We show that directing the edges of the mesh and using this information to assign vertex and edge data to tasks reduces the amount of communication by half. Also, CPE is used to assign tasks associated with grid points to processors to reduce the communication time. We show that using CPE to map tasks to processors reduces the communication time by a factor of 2.23. The result is a load-balanced compute-bound parallel implementation of the Euler code. The performance of this code running on the CM-2 is compared with a similar code running on a Cray-YMP and an Intel iPSC/860. For our test case, we demonstrate that the code on the CM-2 achieves 90% of the performance of the code on 1 processor of a YMP.

In Chapter 5 we compare three methods for computing massively parallel sparse matrix-vector multiplication $y = Ax$. The three methods are: scan-based, column-wise, and row-wise. The scan-based technique stores one nonzero element of A in each processor and uses scan operations to sum locally computed products. The column-wise scheme stores the nonzero elements of a column of A in each processor and computes the sum of the scaled columns. The row-wise method stores nonzero elements of the rows of A in each processor and computes a sparse inner product. For the last two methods, we associate a task with each column and row of the matrix, respectively. We use CPE to map tasks to processors and show that the communication time is reduced by approximately a factor of two for our test cases. We demonstrate that, on the CM-2, row-wise sparse matrix-vector multiplication mapped with CPE is an order-of-magnitude faster than scan-based operations for our test cases.

Chapter 6 contains a summary of and future directions for this research.

1.2 Graph Theory

Following Bondy and Murty [14] and Hartsfield and Ringel [35] we use the following terminology and notation. A *graph* G is a pair of sets (V, E) where V is nonempty, and E is a (possibly empty) set of unordered pairs of elements of V. The elements of V are called the *vertices* of G and the elements of E are called the *edges* of G. We write V_G for the vertices of G and E_G for the edges of G. We use the symbols ν_G and ε_G to denote the number of vertices and edges in G. If there is only one graph being considered, we omit the letter G from the symbols and write, for instance, V, E, ν and ε, instead of V_G, E_G, ν_G and ε_G.

Two graphs G and H are said to be *isomorphic* if there is a bijection $\theta : V_G \rightarrow V_H$ such that $\langle u, v\rangle \in E_G$ if and only if $\langle \theta(u), \theta(v)\rangle \in E_H$. Also, a graph H is a *subgraph* of G if $V_H \subseteq V_G$ and $E_H \subseteq E_G$.

If u and v are vertices of G, we say that u and v are *adjacent* or *neighbors* if $\langle u, v\rangle \in E$. Two edges incident to a common vertex are said to be adjacent. An edge $\langle v, v\rangle \in E$ is called a *loop*.

A *walk* in G is a finite non-null sequence $W = v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, whose terms are alternately vertices and edges, such that, for $1 \leq i \leq k$, the ends of e_i are v_{i-1} and v_i. If the edges $e_1, e_2, \ldots, e_k$ and vertices $v_0, v_1, v_2, \ldots, v_k$ of a walk W are distinct, W is called a *path*. The integer k is the *path-length* of the path. For a graph G, if there is a path from u to v for every u and v in V, then G is said to be *connected*.

With each $\langle u, v\rangle \in E_G$ let there be associated an integer $c(\langle u, v\rangle)$, called its *edge weight*; and, with each $v \in V_G$ let there be associated an integer $w(v)$, called its *vertex weight*. Then G, together with these edge and vertex weights, is called a

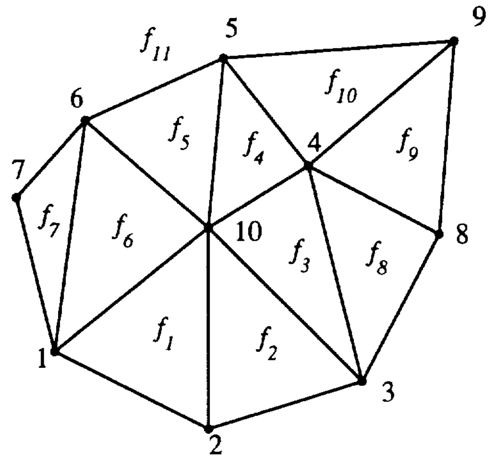


Figure 1.1: A graph G with 11 vertices.

weighted graph. We adopt the convention that $c(\langle u, v\rangle) = \infty$ if $\langle u, v\rangle \notin E$. If H is a subgraph of a weighted graph, the *edge weight* $c(H)$ of H is the sum of the weights $\sum_{e \in E_H} c(e)$ of its edges. We refer to the edge weight of a path in a weighted graph as its *length.* Similarly, the minimum edge weight of a $\langle u, v\rangle$*-path* will be called the *distance* between u and v and denoted by $d(u, v)$.

The *degree* $\delta_G(v)$ of a vertex v in G is the number of edges of G incident to v. Let $\Delta(G)$ denote the maximum degree of the vertices of G and $\delta(G)$ denote the minimum degree of the vertices of G. Figure 1.1 shows a graph G where $\nu = 10$, $\varepsilon = 19$, $\delta = 2$, and $\Delta = 6$.

To any graph G there corresponds an *adjacency matrix*; this is the $\nu \times \nu$ matrix $\mathbf{A}(G) = [a_{ij}]$, where a_{ij} is the number of edges joining v_i and v_j. We define the *Laplacian matrix* of a graph G as $\mathbf{L}(G) = [l_{ij}]$, where $l_{ij} = a_{ij}$ for $i \neq j$ and $l_{ii} = -\delta_G(v_i)$ for each $v_i \in V$. Figure 1.2 shows the adjacency matrix and the Laplacian matrix of the graph G from Figure 1.1.

A graph is said to be *planar* if it can be drawn in the plane so that its edges intersect only at their endpoints. A planar graph G partitions the rest of the plane

$$\mathbf{A}(G) = \begin{bmatrix} 0&1&0&0&0&1&1&0&0&1\\ 1&0&1&0&0&0&0&0&0&1\\ 0&1&0&1&0&0&0&1&0&1\\ 0&0&1&0&1&0&0&1&1&1\\ 0&0&0&1&0&1&0&0&1&1\\ 1&0&0&0&1&0&1&0&0&1\\ 1&0&0&0&0&1&0&0&0&0\\ 0&0&1&1&0&0&0&0&1&0\\ 0&0&0&1&1&0&0&1&0&0\\ 1&1&1&1&1&1&0&0&0&0 \end{bmatrix} \qquad \mathbf{L}(G) = \begin{bmatrix} -4&1&0&0&0&1&1&0&0&1\\ 1&-3&1&0&0&0&0&0&0&1\\ 0&1&-4&1&0&0&0&1&0&1\\ 0&0&1&-5&1&0&0&1&1&1\\ 0&0&0&1&-4&1&0&0&1&1\\ 1&0&0&0&1&-4&1&0&0&1\\ 1&0&0&0&0&1&-2&0&0&0\\ 0&0&1&1&0&0&0&-3&1&0\\ 0&0&0&1&1&0&0&1&-3&0\\ 1&1&1&1&1&1&0&0&0&-6 \end{bmatrix}$$

Figure 1.2: Adjacency matrix and Laplacian matrix of graph G.

into a number of connected regions; the closure of these regions are called the *faces* of G. We shall denote by $F(G)$ and $\phi(G)$, respectively, the set of faces and the number of faces of a planar graph G. For $f \in F(G)$, let $\delta_G(f)$ denote the number of edges incident to f. Finally, let $F_b(G) \subseteq F(G)$ be defined by

$$F_b(G) = \{f \in F(G) \mid \delta_G(f) > 3\}. \tag{1.1}$$

Euler's formula relates the numbers of vertices, edges, and faces of a planar connected graph:

$$\nu - \varepsilon + \phi = 2. \tag{1.2}$$

In Figure 1.1, G is a planar graph, $\phi(G) = 11$, $\delta_G(f_5) = 3$, $\delta_G(f_{11}) = 8$, and $F_b(G) = \{f_{11}\}$. Also, using (1.2) we verify that

$$10 - 19 + 11 = 2.$$

A *hypercube* of dimension n (or n-cube) is a graph $G = (E, V)$ where $\nu = 2^n$ and the vertices are labeled 0 to 2^n-1. There is an edge $\langle u, v\rangle \in E$ if and only if the binary representation of the labelings for u and v differ in exactly one bit position. A *k-ary n-cube* is an interconnection topology with k^n processors. Each processor has an n-digit radix k address, $p = p_{n-1}, \ldots, p_0$. Two processors are adjacent if

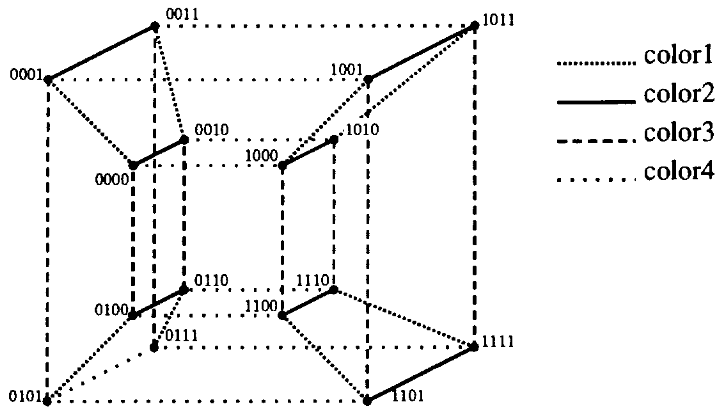


Figure 1.3: Four coloring a 4-cube.

and only if their addresses in radix k differ in precisely one digit, and the magnitude of this difference is one. A hypercube is a k-ary n-cube with $k = 2$.

A *k-edge-coloring* of a loopless graph G is an assignment of k colors, $1, 2, \ldots, k$ to the edges of G. Let E_G^i be the set of edges that are assigned color i. The coloring is *proper* if no two adjacent edges have the same color. A proper k-edge-coloring results in k pairwise disjoint subsets of E_G, such that $\bigcup_{i=1}^{k} E_G^i = E_G$. The *edge chromatic number* $\chi'(G)$, of a loopless graph G, is the minimum k for which G is properly k-edge-colorable. Figure 1.3 shows a 4-cube. The vertex labels are given in binary representation. The graph has a proper 4-edge-coloring and different line types are used to represent the four edge colors. A systematic way of properly k-edge-coloring a k-cube is as follows. If vertices u and v are neighbors and their binary representations differ in the i^{th} position, color edge $\langle u, v\rangle$ with color i. For example, vertices 0000 and 0001 are neighbors and differ in the 1^{st} bit position. Therefore, color 1 is used for edge $\langle 0000, 0001\rangle$.

Let $G = (V, E)$ be a weighted graph and let p be a positive integer. A *p-way partition* of V is a set $\mathcal{P} = \{V_1, \ldots, V_p\}$ of nonempty, pairwise disjoint subsets of V,

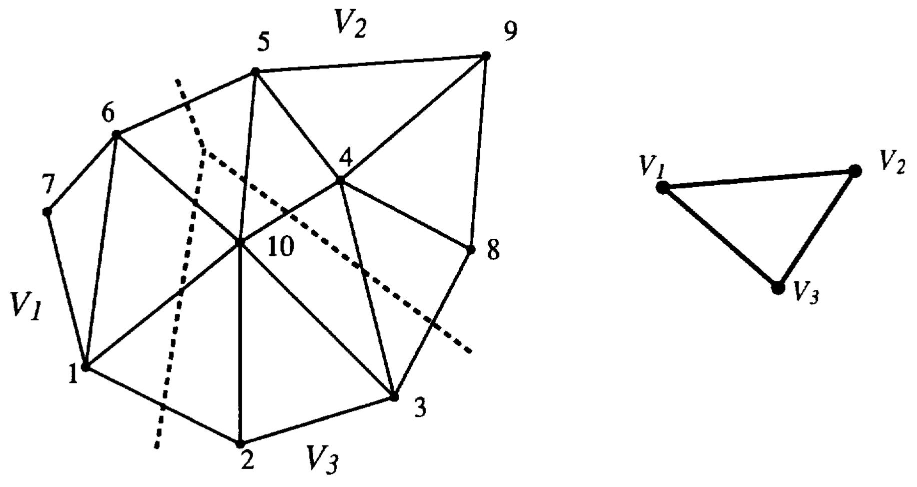


Figure 1.4: A 3-way partition of G and the associated quotient graph.

such that $\bigcup_{i=1}^{p} V_i = V$. Let $q > \max_{v \in V} w(v)$. A partition is *q-admissible* if $\sum_{v \in V_i} w(v) \leq q$ for $i = 1, \ldots, p$. The *partition-cost* of a partition is the summation of $c(\langle u,v \rangle)$ for every $\langle u,v \rangle \in E$ such that u and v are in different subsets. The partition-cost is thus the weighted sum of inter-subset edges. We define the *quotient graph* of G with respect to $\mathcal{P}$ to be the graph $Q_G = (\mathcal{P}, \mathcal{E})$ where $\langle V_i, V_j \rangle \in \mathcal{E}$ if and only if $\langle u,v \rangle \in E$ for some $u \in V_i$ and $v \in V_j$. Figure 1.4 shows a 3-way partition of G and the associated quotient graph. The partitions are $V_1 = \{1,6,7\}$, $V_2 = \{4,5,8,9\}$, and $V_3 = \{2,3,10\}$. The quotient graph $Q_G = (\mathcal{P}, \mathcal{E})$ with respect to this 3-way partition is $\mathcal{P} = \{V_1, V_2, V_3\}$ and $\mathcal{E} = \{\langle V_1, V_2 \rangle, \langle V_1, V_3 \rangle, \langle V_2, V_3 \rangle\}$.

Following Ho and Johnson [38], an *embedding* of a graph G in a graph H is a one-to-one mapping $\varphi : G \rightarrow H$ of vertices of G to vertices of H and an assignment of each edge $e = \langle u,v \rangle \in E_H$ to a path $\varphi(e)$ whose end points are $\varphi(u)$ and $\varphi(v)$. The *dilation* of φ denoted Λ_∞ is

$$\Lambda_\infty = \max(d(\varphi(u), \varphi(v))), \text{ for all } \langle u,v \rangle \in E_G. \tag{1.3}$$

Additionally, the *load factor* of an edge $e' \in E_H$ is $\sum_{e \in E_G} \mid \{e'\} \bigcap E_{\varphi(e)} \mid$. The load factor of φ denoted Λ_{lf} is

$$\Lambda_{lf} = \max_{e' \in E_H} \{ \sum_{e \in E_G} \mid \{e'\} \bigcap E_{\varphi(e)} \mid \}. \tag{1.4}$$

This is the maximum number of paths $\phi(e)$ that share an edge in H.

A *directed* graph (*digraph*) $D = (V, E)$ is a graph whose edges are ordered pairs of vertices. With each digraph D we can associate a graph G on the same vertex set; corresponding to each directed edge of D there is an edge of G with the same ends. G is the *underlying graph* of D. For a digraph D, if u and v are vertices such that $\langle u, v \rangle \in E$ then u is said to be the *tail* of the directed edge and v is called the *head* of the directed edge. The *indegree* $\delta_D^-(v)$ of a vertex v in D is the number of directed edges with head v; the *outdegree* $\delta_D^+(v)$ of a vertex v in D is the number of directed edges with tail v. We denote the maximum outdegree of a digraph D by $\Delta^+(D)$.

We define a *network* N to be a weighted digraph with two distinguished subsets of vertices, X and Y, which are are assumed to be disjoint and nonempty. The vertices in X are the *sources* and those in Y are the *sinks* of N. The edge weight c of each edge is a non-negative integer called the *capacity.* A *cutset* in a network N is a set of edges which when removed disconnects the source nodes from the sink nodes. No proper subset of a cutset is a cutset. The *weight* of a cutset is equal to the sum of the capacities of the edges in the cutset. The Max-Flow Min-Cut theorem (Ford and Fulkerson [46]) states that the value of a maximum flow in a network is equal to the weight of a minimum cutset of that network.

1.3 Architecture of the Connection Machine

Many people have considered mapping applications to hypercube networks [72] and other parallel processors. Hypercubes have attracted much attention because of their topological properties [58, 59] and their development into products by several manufacturers. The experiments conducted here are on a Connection Machine CM-2, which is a hypercube.

The Connection Machine CM-2 is a massively parallel single instruction multiple data (SIMD) computer with 64K 1-bit processors [1]. Instructions are broadcast to the processors from a host computer and all processors execute the same operation at the same time with their own local data. A processor can also decide not to store the result of the broadcast instruction based on local values.

The underlying topology of the CM-2 is an 11-dimensional hypercube of *sprint nodes.* A *sprint node* is composed of 32 1-bit processors (two processor chips containing 16 processors each), a Weitek floating point chip and memory. Neighboring sprint nodes are connected by two bi-directional 1-bit paths. Communication between the 32 processors on a sprint node is very inexpensive relative to communication with processors on other sprint nodes. In order to analyze communication time, one ignores the fact that sprint nodes contain 32 processors and focuses on reducing the cost of inter-sprint-node communication. An 8K processor CM-2 has 256 sprint nodes with 16 bi-directional connections each.

Users of the CM-2 do not have to limit the number of processes or tasks to be less than or equal to the number of processors. A mechanism called a *virtual processor* (VP) allows one physical processor to simulate the operation of multiple processors. On the CM-2 this is done by splitting the memory of each processor into equal pieces; each VP gets its own portion. Each physical processor then executes the same instruction once for each VP. This is transparent to the user. The ratio of the number of virtual processors to the number of physical processors is called the *VP ratio.*

[1]An assessment of the Connection Machine can be found in Schreiber [65].

1.3.1 Regular Communications

Communication is much slower than computation on the CM-2. On a 64K CM-2, point-to-point nearest neighbor communication (often referred to as NEWS communication) rates vary between 7.3×10^7 and 1.6×10^9 floating-point words communicated per second, depending on the VP ratio. Levit [49] showed that using the CM-2 assembly language Paris, the realizable peak computation rate is 5.17×10^9 flops (32-bit). Berryman [9] showed that using the router for collision-free, distance 1 communication is approximately eight times slower than NEWS communication.

Many people have shown that applications which can be implemented using strictly regular (nearest neighbor) communications on the Connection Machine achieve rates close to the realizable peak rate [9, 15, 24, 49, 53, 54, 77]. On the other hand, if an algorithm requires general communication between processors (using the router) then there can be 3 orders of magnitude (or greater) difference between realized and peak performance [9, 18].

Fast multiwire NEWS network communications have been developed by Bromley *et al.* [15] and Myczkowski, Bromley, and McGowan [54]. They develop a combined communication and computation primitive for use with tasks associated with rectilinear grids where communication is conducted between grid neighbors. In a single subroutine call, each task is able to receive a specified variable from a task associated with each of its grid neighbors and have those values added together. However, this work does not apply to arbitrary grids.

Although the multiwire NEWS communication is more efficient than simple NEWS communication, the wires of the hypercube are underutilized. Work by Edelman [22] and Johnsson and Ho [41, 42] addresses this for communications in which each processor has a unique piece of data for every other processor. This is called an "all-to-all personalized" broadcast; it occurs in matrix transpose and bit reversal operations.

1.3.2 Irregular Communications – Communication Compiler

Until recently, general communication on the Connection Machine CM-2 required the router and was excruciatingly slow. This was particularly true for solving unstructured grid problems where the communication pattern required by the application does not match the topology of the CM-2. A feature of the applications that we focus on here is that the communication pattern, although irregular, remains static throughout the duration of the computation. Dahl [19] has developed the *communication compiler*, a software package designed to take advantage of an arbitrary but fixed pattern of communication. Interprocessor communications are scheduled once at the beginning of the program and this schedule is used repeatedly during the program execution. Moving data between sprint nodes using the communication compiler is a factor of 5 to 10 faster than using the router for general communication. An intelligent mapping of tasks to processors as proposed here results in further improvements.

An important concept in compiled communication on the CM-2 is a *message cycle*, a single communication step in which two 32-bit words can be moved across each of the bi-directional paths (one in each direction) connecting sprint nodes. In one message cycle each sprint node on an 8K CM-2 can send and receive 16 words, one for each of its 16 bi-directional connections, for a total of 4096 words per message cycle. The output of the communication compiler is a schedule of data being communicated across wires in time and the number of messages cycles required for the communication to be completed. We define *bandwidth* to be the number of 32-bit words a computer can communicate each cycle. Therefore, the bandwidth of an 8K CM-2 is 4096 words/cycle.

The time to send one or more 32-bit words on the CM-2 using version 6.0 of the communication compiler [20] is given by:

$$time_{CM} = (26 + \underline{51 msg\_cycles} + VPratio(17src + 57rcv))\ \mu sec. \qquad (1.5)$$

The startup cost is 26 μsec. The quantity *msg_cycles* is the number of message cycles for all words to arrive at their destination; *src* is the maximum number of unique 32-bit words being sent from any processor; *rcv* is the maximum number of unique 32-bit words being received by any processor[2].

Throughout this thesis we make use of two communication primitives, *one-to-many* and *many-to-many.* A *one-to-many* communication is when each task has one piece of data to send to a small subset of the other tasks. A *many-to-many* communication is when each task sends a unique word to each member of a small subset of the tasks. In many applications, the subsets of the tasks with which each task communicates are not known until run time because this is determined by data computed at the beginning of the computation or read in as initial data. However, the subsets often remain static for the duration of the computation.

1.4 Contributions of this Thesis

We develop the Cyclic Pairwise Exchange heuristic, a parallel iterative mapping algorithm. It is a highly parallel pairwise exchange algorithm in which each processor may exchange the tasks mapped to it with a small subset of the other processors. CPE requires only a small amount of nearest neighbor communication and it is computationally less expensive than global search methods. The objective function that is minimized is the sum of the distances that words must travel. We demonstrate good correlation between the objective function and the communication time on the CM-2. We show that it is better to reduce the sum of the distances that messages travel than the maximum distance any message has to travel.

For very large, very irregular problems arising in 2-D flow around complex multi-component airfoils and 3-D flow around aircraft, the heuristic outperforms methods based on simulated annealing – it requires far less time to do the mapping

[2]We are referring to the 1-bit processors here. Recall that there are 32 of them in each sprint node.

and the results obtained are better. Compared with random and naive mappings, it reduces the communication time twofold, even for realistic, large, highly irregular, and stretched meshes.

We develop an efficient data parallel implementation of Barth and Jespersen's mesh-vertex upwind finite volume scheme for solving the Euler equations on triangular unstructured meshes [5]. An optimal edge direction is used to group vertex and edge data within a task to reduce the amount of communication by 50% in this application. Also, the edge direction produces a load balanced computation.

We compare three methods of massively parallel sparse matrix-vector multiplication: scan-based, column-wise, and row-wise. For the last two techniques, we use CPE to map tasks associated with the columns and rows of the matrix to the processors. We show that mapping with CPE reduces the communication time by a factor of two for these two methods. Also, we demonstrate that the row-wise method is the fastest of the three and that it achieves approximately an order of magnitude greater throughput than the scan-based method for our test cases on the CM-2.

Taken together, these results demonstrate that a judicious assignment of tasks to processors enables data-parallel SIMD computers to efficiently solve problems that arise in the solution of discretized PDEs, where the discretizing grid is arbitrary, but static.

CHAPTER 2
THE MAPPING PROBLEM

2.1 Problem Statement

A parallel application is composed of many hundreds or thousands of relatively independent tasks. The tasks and their communications form a weighted digraph $T = (V_T, E_T)$ called the *task graph.* There is a vertex $v \in V_T$ for each task and a directed edge $\langle u, v \rangle \in E_T$ if and only if task u sends data to task v. The task graph edge weight $c_T(\langle u, v \rangle)$ is the number of words to be sent from u to v. The vertex weight of T, $w_T(v)$ is the number of floating point operations or instructions executed by task v.

The parallel computer is represented by a connected, weighted graph $P = (V_P, E_P)$ called the *processor graph.* There is a vertex $v \in V_P$ for each processor in the computer. Also, for each processor u that is directly connected with processor v there is an edge $\langle u, v \rangle \in E_P$. The processor graph edge weight $c_P(\langle u, v \rangle)$ is the number of words per second that processor u can send to processor v. The processor graph vertex weight $w_P(v)$ is the number of operations per second that processor v can execute.

Without loss of generality we use the term *task* instead of task graph vertex and *processor* instead of processor graph vertex.

The *mapping problem* consists of finding $\Psi : V_T \rightarrow V_P$ such that we minimize

$$time_{exec} = time_{comm} + time_{comp}. \tag{2.1}$$

Clearly, if Ψ maps all tasks of V_T to one processor, then the communication cost would be zero but the computation time would be high. On the other hand, if there are roughly equal numbers of operations to be done by each task and this is greater than the number of words to be communicated per task, then Ψ should map an equal number of tasks to each processor.

We make the following assumptions about the edge and vertex weights of T and P: the tasks have equal amounts of computations (all w_T are identical), there is an equal amount of data communicated between tasks (all c_T are identical), there is much more computation than communication ($w_T \gg c_T$), and the computer is homogeneous – tasks execute equally well on each processor (all c_P are identical and all w_P are identical).

These are reasonable assumptions since the application areas to which this work applies include computational fluid dynamics, electromagnetics, and structural mechanics. The problems to be solved are nearly always initial or boundary value problems for coupled systems of PDEs. We associate a task with each point in the discretizing grid. The tasks have similar communication and computation requirements and the amount of computation exceeds the communication. For example, in the computational fluid dynamics application discussed in Chapter 4, each task performs approximately 300 floating point operations and communicates 54 words of data for each neighboring task, each iteration of the flow solver [34]. After using CPE, 57.4% of the time is spent computing and 42.6% of the time is spent communicating. Finally, many of the parallel computers being developed and marketed today are networks of homogeneous processors.

Given these assumptions, we choose to approximately solve the problem of minimizing the communication time in a load balanced mapping of tasks to processors: minimize $time_{comm}$ subject to

$$|\{t \in V_T : \Psi(t) = p \in V_P\}| \leq \left\lceil \frac{\nu_T}{\nu_P} \right\rceil, \quad \text{for all } p \in V_P.$$

We approximate $time_{comm}$ with an objective function which is machine independent,

$$\Lambda_1 \equiv \sum_{\langle u,v\rangle \in E_T} c_T(\langle u,v\rangle) d(\Psi(u), \Psi(v)). \tag{2.2}$$

2.2 Complexity

We follow the argument used by Bokhari [12] to show that it is unlikely that an exact polynomial time algorithm exists for solving the mapping problem. If we had an exact algorithm for solving the mapping problem then we could use it to solve the graph isomorphism problem, a classic NP-complete problem. If we make the same assumptions about the mapping problem as above and assume that $\nu_T = \nu_P$, then the mapping problem is identical to the graph isomorphism problem. If we had an exact polynomial time algorithm for the mapping problem, we could use it to map T to P. If the two were isomorphic we would obtain a value of Λ_1 equal to $c(T)$. Thus, we could determine whether or not T and P are isomorphic in polynomial time.

For a mapping algorithm to be practical, less time should be spent on the mapping problem than on solving the application. Typical implicit and explicit schemes for solving discretized PDE's have serial complexity $O(n)$ per iteration, where n is the number of grid points or unknowns. If we associate one task per grid point, then exact or approximate algorithms for the mapping problem should have complexity less than or equal to $O(\nu_T)$, otherwise, more time will be spent computing a mapping than solving the PDE. On the other hand, if the grid remains fixed and the PDE is solved for many different sets of initial values (the task graph remains fixed), then more time can be spent on the mapping problem.

2.3 Taxonomy of Heuristics

Here we describe a taxonomy of the methods for approximately solving the mapping problem. Also, we discuss prior work and where each fits in this taxonomy. Techniques for solving the mapping problem are either heuristic algorithms (heuristics) or exact algorithms. We know that finding an exact or optimal solution to the mapping problem is NP-complete. Thus, any method for finding an exact

```
1. initial assignment
    $S \leftarrow V_T$
    while $S \neq \emptyset$ do
        Choose some $S' \subseteq S$, $S' \neq \emptyset$.
        Assign each $t \in S'$ to a processor.
        $S \leftarrow S \setminus S'$.
    endwhile
2. iterate
    do iterations = 1, maximum number of iterations
        Choose some $T' \subseteq V_T$, $T' \neq \emptyset$.
        Tentatively reassign each $t \in T'$ to a new processor.
        if (new mapping is satisfactory) then
            replace mapping from prev. iteration with new mapping.
        if (mapping is acceptable) then   stop
    enddo
```

Figure 2.1: Outline of Heuristic Algorithms.

solution will almost certainly require an inordinate amount of computation. Heuristics attempt to find an acceptable sub-optimal solution in a reasonable amount of time. A heuristic is a combination of three things: (1) an initial guess at the solution; (2) some improvement procedure; (3) an objective function. The improvement procedure of many heuristics follow the algorithm outlined in Figure 2.1.

Heuristics are divided into two types: one-pass and iterative. One-pass algorithms omit step 2. They incrementally assign an unassigned task or group of tasks to processors, until each task is assigned to a processor. Once an assignment is made, it is not changed.

Iterative techniques execute step 2 one or more times. They are either deterministic or probabilistic. Deterministic-iterative algorithms start with an initial solution to the mapping problem and then at each iteration try to reduce the value of an objective function by changing the assignment of tasks to processors. At each iteration, only one better solution is kept for the next iteration. The process stops when one of the following occurs: some acceptable mapping is achieved, the iteration limit has been exceeded, or no further improvement is possible with the given improvement procedure. Deterministic-iterative algorithms stop at the first local minimum of the objective function. Probabilistic-iterative algorithms use random changes to the assignment of tasks to processors. A new mapping is accepted unconditionally if the value of the objective function is reduced. If the new value of the objective function is higher than the value achieved by the mapping at the previous iteration, then the new mapping is accepted with some finite probability. As with deterministic algorithms, only one solution is kept at each iteration.

The heuristics vary in the way that they make an initial assignment of tasks to processors. In step 2, they differ in the number of vertex mappings changed at each iteration and how the change is made each iteration. They also differ in the objective function used.

2.4 Objective Functions

A good objective function for the mapping problem should have good correlation with the value that it is predicting – the execution time of the application – as well as being readily computable to be practical.

In addition to the variety of optimization procedures, there is a spectrum of objective functions one can use in solving the mapping problem. We use subscripted Λ's to denote the various objective functions defined in this section.

One objective function that has been used is Λ_∞ (1.3). Recall that this is the maximum distance between neighbors in the task graph under the mapping. Its advantages are that it is easy to compute and it is a lower bound on the communication time. However, Λ_∞ only measures the maximum distance that data travels;

the total amount of communication is not measured. Experiments on the CM-2 with our test cases show that this objective function is not well correlated to the actual communication time.

Bokhari [12] suggests using the *B-cardinality* of the mapping:

$$\Lambda_B \equiv \sum_{\langle u,v\rangle \in E_T} c_P(\langle \Psi(u), \Psi(v)\rangle), \tag{2.3}$$

with the assumption that $c_P(e) = 1$ if $e \in E_P$ and $c_P(e) = 0$ otherwise. Unlike other objective functions discussed here, Λ_B is to be maximized. It counts the number of neighboring task graph vertices that are mapped to neighboring vertices in the processor graph. It only accounts for task graph edges that are mapped to length one paths in the processor graph. However, task graph edges mapped to paths of length greater than one can be the communication bottleneck; they are not accounted for in this objective function.

Sadayappan *et al.* [61] suggest minimizing:

$$\Lambda_p \equiv \sum_{\langle u,v\rangle \in E_T} \begin{cases} 1, & \text{if } d(\Psi(u), \Psi(v)) \geq 1 \\ 0, & \text{otherwise} \end{cases} . \tag{2.4}$$

This is the number of adjacent vertices in the task graph that are mapped to processor graph vertices which are separated by a distance of one or greater. This objective function is also easy to compute but does not differentiate between short and long distance communication. Also, it does not account for the amount of communication represented by a task graph edge.

We approximate $time_{comm}$ of (2.1) with Λ_1 of (2.2). The objective function Λ_1 is both readily computed and it is easily parallelized. Also, we find that Λ_1 is a good approximation to the actual communication time on the CM-2 even though it does not explicitly account for contention on communication paths. In Section 3.5 we show that there is good correlation between Λ_1 and communication time on the CM-2.

One can also use Λ_{lf} (1.4) as an objective function to be minimized. First, a p-way partition of the task graph is made and then the associated quotient graph is embedded in the processor graph, minimizing the load factor. This is a better approximation to the actual communication time than Λ_1 but it is much more expensive to compute since edges in the quotient graph must be assigned to paths in the processor graph. This requires one to solve a series of maximal bipartite matching problems to optimally schedule the communications.

Lee and Aggarwal [48] advocate using the number of message cycles (they call it the communication overhead) as an objective function, denoted Λ_{mc}. They schedule task graph edges to paths in P and determine which edge will take the maximum number of message cycles to complete its associated communication. Both the length of the paths and the contention for processor graph edges are measured by Λ_{mc}. Although Λ_{mc} is a very accurate measure of the communication time, it is expensive to compute each time one considers changing the mapping since computing an optimal schedule is itself a hard combinatorial optimization problem.

2.5 Prior Work

Graph partitioning and *graph embedding* are related to graph mapping. A mapping is both a partitioning and an embedding. The subsets of V_T consisting of the tasks mapped to the same processor constitute a partition of V_T. To approximately solve the mapping problem, one might consider first partitioning V_T into ν_P disjoint subsets and then embedding the quotient graph into the processor graph (ν_P processors). Graph partitioning followed by embedding can be viewed as a one-pass heuristic or as an initial mapping to be improved by an iterative heuristic.

2.5.1 Graph Partitioning

The *graph partitioning problem* is: given a weighted graph G and positive integers p and q, find a q-admissible p-way partition of G with the lowest partition-cost. Since graph partitioning is known to be NP-complete [29], heuristics have been used to find acceptable solutions.

Kernighan and Lin [43] consider the case where all $w_G = 1$. They allow the edges to be weighted. Their work is motivated by two applications; placing the components of an electrical circuit onto printed circuit boards so as to minimize connections between cards, and improving the paging properties of computer programs by assigning subroutines, procedure blocks, data items, etc., to pages of memory so as to minimize the references to objects that reside on different pages. They first consider 2-way partitions. An initial partition is made and then repeated pairwise exchanges are made to improve the initial partition. Empirically, they determine that the time complexity of this heuristic for finding a 2-way partition in a graph with ν vertices is $O(\nu^2)$. A feature of this work is that sequences of perturbations are considered rather than single perturbations which endows the method with some ability to bypass local minima. This is superior to other simple local heuristics. Finally, k-way partitions are made using repeated application of the 2-way procedure.

Fiduccia and Mattheyses [25] improve the Kernighan-Lin work. They use efficient data structures and vertex displacements instead of exchanges to derive a linear time heuristic for improving 2-way graph partitions.

Gilbert and Zmijewski [32] develop a parallel version of the Kernighan-Lin algorithm to find low cost partitions for factorization of sparse matrices. These partitions are then used to compute orderings for factoring matrices. In addition to reducing fill, the resulting orderings lead to good processor utilization and low communication overhead. The computational complexity of the algorithm they develop is $O(\varepsilon_T \log \nu_T \log \nu_P)$.

```
To find a $2^n$-way partition in $G$.
Set $\mathcal{P}_0^0 = V_G$
**do** $i = 0, \ldots, n-1$
    **do** $j = 0, 1, \ldots, 2^{i+1} - 1$
        Compute $x_2$ of $\mathcal{P}_j^i$.
        Sort the components of $x_2$.
        Assign half of the vertices and edges corresponding to the smallest com-
          ponents in $x_2$ to $\mathcal{P}_{2j}^{i+1}$ and those corresponding to the other half to
          $\mathcal{P}_{2j+1}^{i+1}$.
    **enddo**
**enddo**
$\mathcal{P}_i^n$, $i = 0, 1, \ldots, 2^n - 1$ are the subgraphs of $G$.
```

Figure 2.2: Recursive Spectral Bipartition (RSB) Algorithm.

Pothen, Simon, and Liou [55] and Simon [70] partition the graphs of sparse matrices using the *spectral method* of Fiedler [26, 27], also called *recursive spectral bipartitioning* (RSB). The RSB method for graph partitioning uses the eigenvector x_2 corresponding to the second largest eigenvalue λ_2 of the Laplacian matrix of the graph to find vertex separators. The largest eigenvalue of $\mathbf{L}(G)$ is zero. If G is connected then λ_2 is negative. If the vertices of a graph are numbered from 1 to ν, then the i^{th} component of an eigenvector corresponds to the i^{th} vertex. The components of x_2 yield a weighting for the corresponding vertices. The differences in these weights give relative distance information about the vertices of the graph. Sorting the vertices according to their weights provides a way to partition G. The algorithm used in [55, 70] to compute a $2^n - way$ partition of a graph G is shown in Figure 2.2. They show that RSB is much better than other methods for finding vertex separators to partition graphs, such as the nested dissection algorithm of George [30].

SA is an optimization technique from statistical mechanics. It simulates the slow cooling of solids to develop efficient methods for finding the extremum values of a function with many independent variables. SA works by iteratively proposing new values of the independent variables and then evaluating the objective function. If the value of the objective function for the new values is less than the value of the objective function for the previous values, then the new values are kept. If the objective function increases, then the new values are kept with some probability. This process is repeated until a desired solution is achieved or a maximum number of iterations is exceeded. SA requires the user to specify parameters of the algorithm – beginning and ending "temperatures" and a "cooling" schedule. However, finding a combination of these that produces a good mapping in a small amount of time is very difficult since they may differ for every task graph and every processor graph.

Savage and Wloka [63] introduce a parallel heuristic for bi-partitioning random graphs. It incorporates some features of the Kernighan-Lin heuristic and some features of simulated annealing (SA) [44]. The Savage-Wloka heuristic randomly groups large numbers of equally good candidates into sets and then deterministically swaps sets between partitions. They call this the *Mob* heuristic and it is implemented on a CM-2. They show that Mob, used on two standard random graph test cases, achieves lower cost partitions than their implementation of the Kernighan-Lin and SA heuristics.

Another approach to solving the partitioning problem is to use a genetic algorithm (GA) – a stochastic search and optimization technique. Talbi and Bessière [75] implement a parallel GA on a mesh of 64 Transputers. They partition task graphs with unequal vertex weights and assume that the processor graph has uniform vertex weights. The objective function Λ_{tb} they minimize is the sum of the inter-partition communication costs plus the variance of the sum of the weights of the tasks assigned to each partition.

```
Generate initial population $S$.
do number-of-generations times
    forall$\sigma \in S$, evaluate $\Lambda_{tb}$
    Select pairs of individuals to reproduce.
    Apply genetic operators to pairs selected to reproduce.
    Eliminate $\sigma$'s with largest $\Lambda_{tb}$ , keep population size constant.
enddo
Choose best individual from the population as the solution.
```

Figure 2.3: A Typical Genetic Algorithm.

To find a ν_P-way partition in a task graph using a GA heuristic, the search space (of size $\nu_P^{\nu_T}$) is the set of ν_T-vectors with components in V_P. A GA starts with a set of ν_T-vectors called the *initial population*, which is typically generated randomly. Each member of the population is called an *individual.* A set of genetic operators is used to generate new individuals from the initial or previous population using a process called *reproduction.* During reproduction, some of the individuals are replaced keeping the size of the population fixed. The basis of GAs is as follows: the closer an individual comes to minimizing the objective function, the more likely it is to reproduce. Two common forms of reproduction in a GA are *crossover* and *mutation.* Crossover is the process of splitting two individuals (n-vectors) at the same random location and exchanging corresponding sections to make two new individuals. Mutation is the process of changing a randomly selected element or elements of an individual to a randomly selected V_P. The probability that an individual selected for reproduction goes through crossover or mutation is a tunable parameter which can change from iteration to iteration. A typical GA is shown in Figure 2.3.

In [75], the GA is parallelized by having a population of 64 individuals each assigned to one of the 64 processors. Evaluation of f for each individual is done in parallel. Each σ assigned to processor p considers a subset of the population for reproduction. A possible subset are the σ's assigned to processors that are directly connected to p. Other subsets are possible, this is another tunable parameter.

Talbi and Bessière compare their parallel GA with serial implementations of a hill-climbing heuristic and a SA heuristic for two test problems: find an 8-way partition in a task graph with 32 vertices and find a 4-way partition in a task graph with 64 vertices. They show that using their GA algorithm on the two test cases results in lower values of the objective function than when they use their hill-climbing and SA heuristics.

The problem with using a GA is that there are many parameters that the user must set: (1) the probability that an individual will go through crossover, (2) the probability that an individual will mutate, and (3) the subset of the population with which each individual can reproduce. Finally, the user sets the order that the $t \in V_T$ appear in the n-vector representing each individual in the population. The order is important because crossover exchanges groups of contiguous elements. The entries representing tasks that reside near each other in an n-vector are likely to be changed together. A good choice of these parameters can result in a GA producing a good mapping. However, such a combination is very difficult to find since they may differ for every task graph and every processor graph.

2.5.2 Graph Embedding

One problem that has been studied is whether a graph T can be embedded into graph a P such that $\Lambda_\infty = 1$. This is equivalent to asking if T is a subgraph of P. The general problem is NP-complete. It has long been known that multi-dimensional grids of suitable dimension can be embedded as subgraphs of the hypercube by means of gray codes [31]. Independently, Krumme, Venkataraman, and Cybenko [45] and

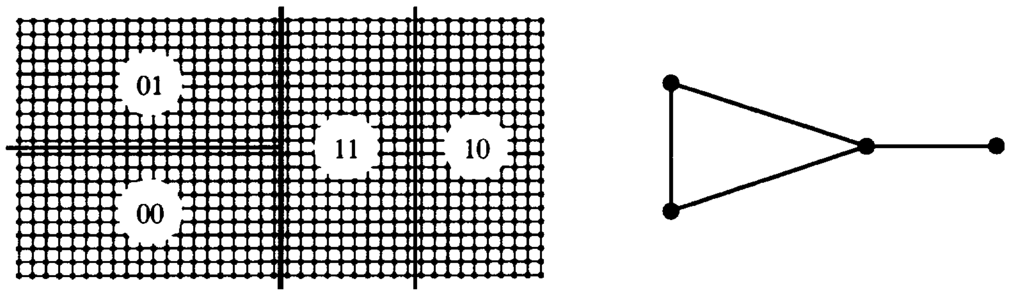
Afrati, Papadimitriou, and Papadimitriou [1] show that the problem of deciding whether an arbitrary graph is a subgraph of a hypercube is NP-complete.

Bhatt and Ipsen [10] show that a complete binary tree with $2^n - 1$ nodes can be embedded in a 2^n node hypercube with $\Lambda_\infty = 1$ everywhere except one edge which has dilation 2. Chan and Chin [16] consider embedding 2D grids into hypercubes with at least as many nodes as grid points. They develop an embedding scheme for an infinite class of 2D grids such that $\Lambda_\infty \leq 2$.

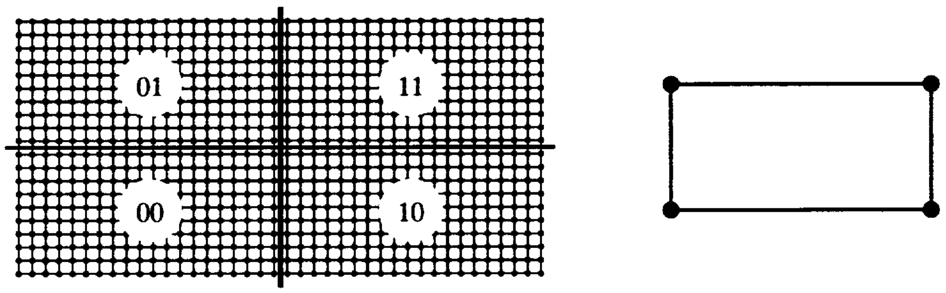
Savage and Wloka [64] embed large random graphs in grids and hypercubes using their Mob heuristic [63] on a CM-2. It exchanges the assignment of large sets of vertices at once. They show that for two standard random graph test cases, they achieve lower cost embeddings than a heuristic based on SA.

2.5.3 Ercal's Example

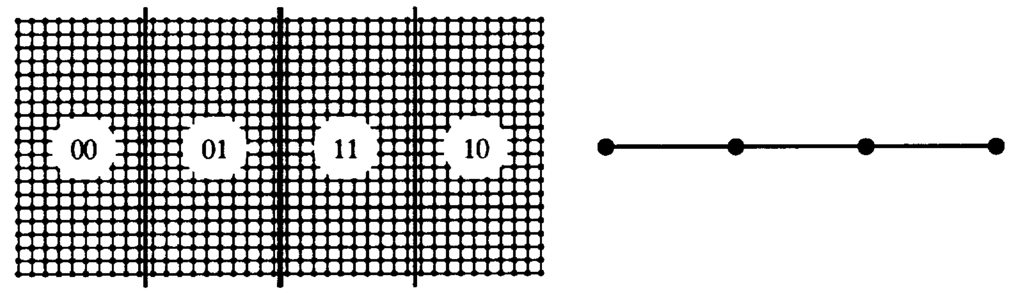
As mentioned above, one can first partition the task graph and then embed the partitioned graph into the processor graph. Ercal *et al.* [23] argue that this is not a good approach to solving the mapping problem. They claim that performing the two operations in isolation can lead to poor mappings and much less than optimal communication time. Also, they say that graph partitioning techniques based solely on minimizing the number of edges cut, subject to some load balancing constraints, can make poor choices for partitions. Ercal *et al.* use the following example to illustrate this. Consider mapping the tasks of a 40×20 task graph to a 2-cube (4 processors) by first computing a 4-way partition of the graph. Figure 2.4a-c shows three 4-way partitions of the grid and the quotient graph. The first partition is the wide vertical line splitting the grid into two 20×20 pieces. After the first partition is made, there are 4 possible choices for the second level partitions. Three of these are shown in the figure. The one not shown is the same as shown in Figure 2.4a except that the right partition is made with a horizontal line and the left partition is made



a: 4-way partition and its quotient graph



b: 4-way partition and its quotient graph



c: 4-way partition and its quotient graph

Figure 2.4: Possible 4-way graph partitions, processor mappings, and quotient graphs of a 40×20 grid.

with a vertical line. All four partitions are optimal - the sizes of the partitions are the same and the number of edges cut are the same. However, they are not equal from the embedding perspective. The quotient graph has to be embedded in the system graph. Partitions 2 and 3 can be embedded into a 2-cube with $\Lambda_\infty = 1$ while partition 2.4a (and the similar partition not shown) cannot. The quotient graph in partition 2.4a has an odd length cycle and hypercubes do not. Therefore, embedding the quotient graph of this partition will result in $\Lambda_\infty = 2$. Additionally, embedding partition 2.4a (and its similar partition) into a 2-cube will result in Λ_{lf} ≥ 2. Partitions 2.4b and 2.4c can be embedded into a 2-cube with $\Lambda_{lf} = 1$. A mapping scheme that independently partitions and then embeds has a 50-50 chance of finding an optimal solution to this example mapping problem.

Even though this simple example shows that partitioning followed by embedding can lead to a poor mapping, we show that, in practice, partitioning followed by embedding produces good mappings. In particular, we achieve the best mappings for our test cases by using spectral partitioning and embedding the quotient graph into the into the processor graph to form an initial mapping and then using CPE to further improve the mapping. Also, graph partitionings that minimize the partition-cost are well-suited for parallel computers with high latency, because the communication time due to the distance between processors is negligible.

2.5.4 Mapping Problem

We now discuss prior work on the mapping problem. We highlight the most significant contributions to the area. They are presented in chronological order.

Stone [73] develops a one-pass heuristic for the mapping problem. He uses a network flow algorithm as a "black box" utility to map a task graph to a two-processor system. A *network* representation of the mapping problem is constructed and fed to a network flow algorithm.

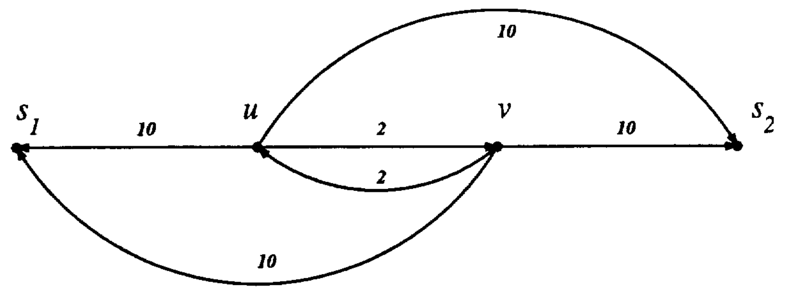


Figure 2.5: A network flow graph constructed from a task graph with two vertices.

The construction of a network representation N of the two-processor mapping problem is as follows:

1. $N = T$.
2. Add nodes labeled s_1 and s_2 to V_N representing the two processors. s_1 is the unique source and s_2 is the unique sink.
3. For each $v \in V_T$, add an edge $\langle v, s_1 \rangle$ and $\langle v, s_2 \rangle$ to E_N.
4. Let $c(\langle v, s_1 \rangle)$ be the estimated time to execute task v on processor s_2 and $c(\langle v, s_2 \rangle)$ be the estimated time to execute task v on processor s_1.

The edge weights are chosen so that the weight of a cutset of N is equal to the execution time of the corresponding task-to-processor mapping. An optimal mapping of tasks to two processors is found by finding a minimum weight cutset, and assigning tasks to the processor on the same side of the cut.

For example, in Figure 2.5 we show a task graph $T = (V_T, E_T)$ where $V_T = \{u, v\}$ and $E_T = \{\langle u, v \rangle, \langle v, u \rangle\}$. The edge weights are $c(\langle u, v \rangle) = 2$ and $c(\langle v, u \rangle) = 2$. A network is constructed by adding the vertices s_1 and s_2 and edges $\langle u, s_1 \rangle$, $\langle u, s_2 \rangle$,

$\langle v, s_1\rangle$, and $\langle v, s_2\rangle$ with weights $c(\langle u, s_1\rangle) = 10$, $c(\langle u, s_2\rangle) = 10$, $c(\langle v, s_1\rangle) = 10$, and $c(\langle v, s_2\rangle) = 10$. Therefore, the tasks execute equally well on either processor and five times as much time will be spent computing as communicating.

Network flow algorithms do not always provide good solutions to the mapping problem. In the example above, the maximum flow algorithm assigns both u and v to the same processor since there are two minimum weight cutsets with weight 20, $\{\langle u, s_1\rangle, \langle v, s_1\rangle\}$ and $\{\langle u, s_2\rangle, \langle v, s_2\rangle\}$. Putting both tasks on one processor results in a running time of 20; one only task computes at a time. The communication time is zero since both tasks are in the same processor. However, if the computations can be done in parallel and the communications are completed serially, then the running time when the tasks are mapped to different processors is 14. Constructing a network in this manner does not account for the concurrency in the two tasks. The result is a mapping that requires more execution time than if the tasks were mapped to different processors. Additionally, using a network flow based heuristic to solve the mapping problem is computationally expensive. Efficient Max-Flow Min-Cut algorithms are of complexity $O(\varepsilon_N \nu_N \log \nu_N)$ [28].

Stone generalizes this approach to ν_P-processor networks although he does not give a complete efficient algorithm. He shows that a single source network flow algorithm can give information about the minimal weight cutset in a ν_P-processor graph. Let $S = \{s_1, \ldots, s_\nu\}$ be the distinguished nodes representing ν_P processors. For $i = 1, \ldots, \nu_P$, run a single source network flow algorithm using s_i as the source node and $S \setminus s_i$ sinks. Stone proves that if some v is associated with s_i by the two-processor flow algorithm then v is associated with s_i in a minimum cost cutset in a ν_P-processor network. Unfortunately, one can construct examples in which some v is mapped to a processor in the ν_P-processor cutset, but fails to be associated with that processor by the two-processor cutset. Therefore, even after ν_P applications of the two-processor network flow algorithm, some subset of V_N may not be mapped to a processor.

Bokhari [12] develops a combined deterministic and probabilistic, iterative heuristic. The objective function he uses is Λ_B. The heuristic consists of pairwise exchanges that attempt to maximize Λ_B. First, an initial assignment of tasks to processors is made. Next, the heuristic loops for each of the ν_T tasks:

1. Consider swapping the mapping of this task with the mapping of all other tasks.

2. Exchange processor assignments between the pair that leads to the largest increase in Λ_B.

In this inner loop, only one pair of vertices can change their mapping at each iteration. If at least one exchange is made through the loop over all tasks, the loop is repeated. If no exchange is made, the current mapping is saved and a random jump to a nearby mapping is made by permuting the mappings of $\sqrt{\nu_T}$ randomly selected tasks. If the new (permuted) mapping has a smaller value of Λ_B than the saved mapping, then the saved mapping is kept and the heuristic stops. If the new mapping has a larger value of Λ_B than the saved mapping, the new mapping replaces the saved mapping and the loop is repeated until no further improvement is made. The complexity of the outer iteration for this heuristic is $O(n^2)$. It is not efficient for large problems.

Lee and Aggarwal [48] develop a deterministic-iterative heuristic mapping strategy for parallel processors using an accurate characterization of the communication overhead. Their target machine is the hypercube and they assume $\nu_T = \nu_P$. They introduce three objective functions to evaluate the quality of a mapping. The first objective function is the sum of the message cycles from each task which is appropriate if no two communications occur at the same time. The second objective function is the maximum number of message cycles which is appropriate if all communications occur simultaneously. The third objective function is the sum of the maximum number of message cycles at each stage which is appropriate if the communication occurs at different stages. To evaluate the objective functions one must assign task graph edges to processor graph paths every time the mapping is changed. An initial assignment of tasks to processors is made using a one-pass approach which attempts to match the communication requirements of tasks to the communication capacities of processors. The complexity of the initial mapping is $O(\nu_T^2)$ [48]. Then, they perform serial, pairwise exchanges and evaluate the quality of the mapping using the appropriate objective function. The pairwise exchange used is similar to the one used by Bokhari [12]. The objective function is evaluated for every candidate exchange. The pairwise exchange that results in the largest decrease in the objective function is made.

Berger and Bokhari [7] study mapping refined grids to parallel processors interconnected by a mesh, binary tree, and a hypercube. The task graphs they consider are initially regular grids that are refined by imposing increasingly finer grids over a region of the global coarser grid. They use Λ_B as the objective function and a one-pass algorithm to map tasks to processors. The task graphs are partitioned into load-balanced, disjoint subgraphs by recursive orthogonal bisection (ROB). ROB recursively partitions a planar graph by placing a horizontal or vertical line such that half the vertices lie on either side of it. Each half is then bisected in the same manner by a line orthogonal to the previous partitioning line. This is done recursively until the number of partitions matches the number of processors. The partitions are then embedded in the processor graph. They achieve lower cost mappings on the hypercubes and meshes than on the binary tree interconnection schemes. However, the results for the hypercube are only marginally better than for the mesh. No experiments are performed to show that communication time on a hypercube, mesh, or binary tree connected multiprocessor decrease as predicted by the Λ_B.

Berman and Snyder [8] study the graph mapping problem and use context-free grammars to generate a class of task graphs. The class includes complete binary trees, cube-connected cycles, hex and square meshes, toruses, linear and multidimensional arrays, butterflies, and complete graphs. The task graphs are then partitioned and embedded in the processor graphs. The partitioning is done by "contracting" the task graph into a similar graph (one also generated by the same context free grammar) with fewer vertices. This is done repeatedly until the contracted task graph and the processor graph have an equal number of vertices. This work does not apply to the unstructured grids considered here.

Ercal *et al.* [23] map finite element grids with several hundred grid points to a hypercube using a one-pass heuristic. They compare SA to a recursive bisection method based on the Kernighan-Lin work and use Λ_p as the objective function. They show that, on average, their recursive bisection scheme reduces the objective function almost as well as the SA approach, but requires approximately two orders of magnitude less time to achieve the results. No timings were made utilizing these schemes in an application running on a hypercube.

Sadayappan *et al.* [60] compare two one-pass heuristic algorithms for mapping regular grids to hypercubes: a cluster mapping and a nearest neighbor mapping. The cluster mapping is based on the Kernighan-Lin algorithm. The nearest neighbor mapping requires a regular grid in order to be effective. They show that cluster mapping is more effective than the nearest neighbor approach at reducing communication time for systems with high message startup costs. Neither approach produces parallel efficiency greater than 50% when mapping irregular grids to a 16-processor iPSC/1. No mention is made of how they embed the quotient graph.

Williams [81] compares three parallel partitioning techniques for graph mapping: SA, ROB, and RSB. He assumes that interprocessor communication time is independent of the distance between processors and does not give any details about how the quotient graph is embedded into the processor graph. The implementation is on a 16-processor NCUBE machine. In his experiments, execution time of ROB is less than the execution time of RSB. The parallel implementation of SA takes 20 times longer to run than RSB for his test cases. Finally, the running time of an application is measured after being mapped by the three methods. For a task graph of 5772 nodes, the running time of the application is fastest for SA and slowest for ROB. Even though SA ran significantly longer than ROB, the running time for the SA-partitioned application (best mapping) was 21% less than the running time for the ROB-partitioned application (worst mapping).

Dahl [19] develops a parallel implementation of SA on the CM-2. He shows that it is effective at reducing Λ and the communication time for the class of problems considered here. It is attractive because it can achieve good results if run long enough and typically avoids getting stuck in local minima. However, as was mentioned before, it requires the user to specify parameters of the algorithm. Like the GA used for graph partitioning, optimal choice of these parameters differ for every graph and if chosen incorrectly can greatly increase the running time of the heuristic. In Chapter 3, we compare Dahl's implementation of SA and CPE for several test cases.

Saltz *et al.* [62] use ROB to map two unstructured task graphs arising in computational fluid dynamics. They compare the communication time for each task graph after using a ROB and a gray code initial mapping. They define a *gray code* mapping to be the assignment of task t to the processor whose number is the binary gray code of t. But, since the numbers assigned to tasks are not done in any particular order, this can be a poor mapping. Compared to their gray code mapping, ROB reduces the communication time by a factor of 2.31 and 4.75 on an 8K processor CM-2 for their two test cases.

Search techniques are exact algorithms that have been used to solve the mapping problem. Shen [68] considers an optimal assignment in which communicating modules are required to reside in the same or neighboring processors. An A^* search algorithm (an ordered search of a general state space graph) is used to search the space of feasible assignments. Sinclair [71] uses a state space reduction technique (branch-and-bound-with-underestimates) to find optimal embeddings. The drawback of search techniques is that for ν_T tasks and ν_P processors there are $(\nu_T)^{\nu_P}$ possible assignments of tasks to processors. Searching is not feasible for large problems since in the worst case it requires a complete enumeration of all possibilities and this is prohibitively expensive for large ν_T and ν_P.

Work on the mapping problem has also been done by others. Reed *et al.* [56] and Lee [47] study the mapping of regular grid problems arising in finite difference equations to hypercubes. They show that non-rectangular (e.g., triangles, diamonds, and hexagons) partitions are better for some discretization stencils than rectangular partitions.

Schwan *et al.* [66] develops a one-pass algorithm to map regular grid problems to hypercubes. He assigns horizontal or vertical strips of the grid to processors in the hypercube. This approach is limited to regular grids.

Lo [50] extends Stone's network approach. Recall that in a homogeneous system, an optimal network solution will map all tasks to one processor. She adds a penalty function to distribute the tasks to multiple processors. However, repeated use of the Max-Flow Min-Cut algorithm is too expensive for this approach to be practical for large problems.

Yalamanchili and Lee [82] use SA to solve the mapping problem. For their tests, they map a 4-cube to a 4-cube, a 5-cube to a 3-cube, a regular mesh of 16 tasks to a mesh connected processor with 16 nodes, and 16 tasks interconnected in a 2-D torus to a 4-cube. A random initial mapping of tasks to processors is used and 50 trials are run for each test case. The optimal mapping is achieved in 90% of the trials for each test case.

André *et al.* [3] compare the performance of four mapping algorithms. They implement the pairwise exchange algorithm of Bokhari, an SA algorithm, and two one-pass algorithms. They use Λ_B as the objective function. The first one-pass algorithm assigns tasks to processors one by one, assigning the unassigned task that optimizes the objective function at each step. The second one-pass algorithm works similarly except that it assigns the unassigned task with the highest communication load with previously assigned tasks at each step. The following task graphs are mapped to a 4-cube: a 4×4 grid, a complete binary tree of 15 nodes, and a 4-cube. They achieve optimal or near-optimal mappings for these small test problems using the algorithms. However, SA requires a factor of 50 more time than the others to achieve the same results.

2.6 Summary of Prior Work

The prior work described here demonstrates that heuristics can effectively reduce the communication time of parallel applications by assigning communicating tasks to nearby processors. They also show that it is important to have an objective function that can be quickly evaluated and that accurately predicts the communication time of an application on a parallel computer. Although several different objective functions were introduced above, it has not been shown that a reduction in these objective functions produced a commensurate reduction in the communication/execution time of an application. Lee and Aggarwal [48] advocate using the actual communication overhead but, in practice, this is difficult to obtain.

Another open question concerns the efficiency of the heuristics that have been developed to date. All of the deterministic-iterative heuristics are variants of the pairwise exchange algorithm developed by Bokhari [12]. It requires $O((\nu_T)^2)$ operations per loop since one tries to exchange the mapping of each task with the mapping of every other task. This requires global information which is inefficiently gathered on massively parallel computers; the processors are typically sparsely connected. In the next chapter we develop the Cyclic Pairwise Exchange heuristic which has serial complexity $O(\nu_T)$. One considers exchanging the mapping of each task with a small set of other tasks assigned to neighboring processors.

CHAPTER 3
THE HEURISTIC

3.1 Description of CPE

In this chapter we discuss Cyclic Pairwise Exchange (CPE) and how it is used to find the mapping $\Psi : V_T \rightarrow V_P$ that reduces Λ_1 and thus the communication time. Also, we show that reducing the number of message cycles produces a commensurate reduction in the communication time and we show that there is good correlation between Λ_1 and the number of message cycles.

Recall from (2.2) that

$$\Lambda_1 = \sum_{\langle u,v\rangle \in E_T} c_T(\langle u,v\rangle) d(\Psi(u), \Psi(v)).$$

CPE starts with an initial assignment of tasks to processors and a proper k-edge-coloring of the processor graph P. It loops over the edge colors, iteratively improving the mapping by performing, in parallel, pairwise exchanges of the tasks among processors connected by an edge of the selected color. We call one loop over all k colors a *sweep*.

For $u \in V_T$ and $q \in V_P$, define

$$\lambda(u,q) = \sum_{v \in adj(u)} c_T(\langle u,v\rangle) d(q, \Psi(v)).$$

It is the weighted sum of the distances that all data originating at a single vertex u must travel under the mapping Ψ restricted to $V_T \backslash u$, if u is mapped to processor q. Also, let $reduc(u,q) = \lambda(u, \Psi(u)) - \lambda(u,q)$, the amount that Λ_1 is reduced if u is remapped to processor q and all other task mappings remain unchanged. Finally, for $r, q \in V_P$, $best(r,q)$ is the task u with $\Psi(u) = r$ and $reduc(u,q)$ is maximum, or the task currently mapped to processor r that causes the largest reduction in Λ_1 if it is instead mapped to processor q.

```
do sweeps = 1, max # sweeps
    do $i = 1, \ldots, k$
        forall $q, r \in V_P$ such that $(q, r) \in E_P^i$:
            Compute $best(r, q)$ and $best(q, r)$.
            if $((reduc(best(r, q), q) + reduc(best(q, r), r)) \geq 0)$ then
                Exchange mappings of $best(r, q)$ and $best(q, r)$.
            endif
    enddo
    if $(\Lambda_1 \leq$ tolerance) stop
enddo
```

Figure 3.1: The CPE Algorithm.

The CPE algorithm is shown in Figure 3.1. CPE loops over subsets E_P^i, $i = 1, \ldots, k$. Each edge $\langle r, q \rangle \in E_P^i$ connects a pair of processors and, since the edge coloring is proper, r and q are not adjacent to any other edge in E_P^i. CPE chooses two vertices that are mapped to the pair, one mapped to r and one mapped to q, and tries to exchange their mappings. One pairwise exchange is possible for each edge. The exchange that causes the largest reduction in Λ_1 (if any) is made every iteration. (Each pair independently chooses to make an exchange or not.) Note that an exchange is made if the *sum* of the two reductions is non-negative. Moving one node may contribute negatively, but as long the negative reduction is offset by a positive reduction of sufficient magnitude from the other vertex, the exchange is made.

In each sweep, a task $v \in V_T$ has the opportunity to exchange its position with any task $u \in V_T$, such that $d(\Psi(v), \Psi(u)) = 1$. CPE differs from prior heuristics in several ways. The set of possible exchanges for a vertex $v \in V_T$ is a small subset of V_T rather than all of V_T. Also, the subset is defined by processor graph neighbors of $\Psi(v)$ rather than task graph neighbors of v. This was done so that, in parallel, each

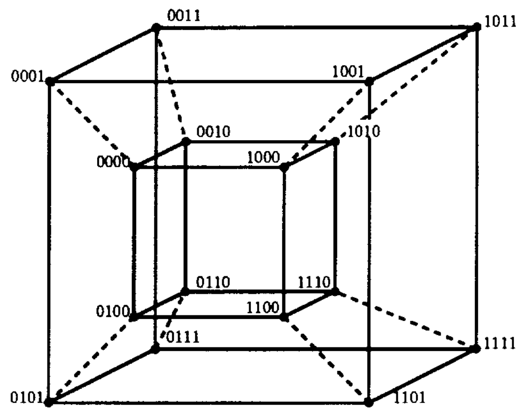


Figure 3.2: A 4-cube with edges of E_P^1 highlighted, splitting the 4-cube into two 3-cubes.

processor could compute the value of *best* and then exchange information with a neighboring processor to perform the pairwise exchange. One can take advantage of fast nearest neighbor communications without using a slower global communication network to perform the exchange. Also, parallel exchanges of ν_P pairs can occur each iteration.

The implementation of CPE on a k-cube, mapping a task graph to a k-cube P, is as follows. For a k-cube, $\chi'(P) = k$. In this minimum edge coloring, an edge $\langle r, q \rangle \in E_P^i$ if and only if the binary representations of r and q differ in only the i^{th} bit position. For each i, the k-cube is partitioned into two $(k-1)$-cubes. Figure 3.2 shows a 4-cube partitioned into two 3-cubes. The dashed lines represent edge color 1 and they highlight pairs of hypercube processors. Note that processors (0001) and (0000) only differ in the 1^{st} bit position and therefore are connected by a dashed line. CPE loops over each of the k edge colors of the hypercube, splitting the cube in half each iteration, and making pairwise exchanges between subcubes. CPE is implemented similarly on 2-D and 3-D grids and 2-D and 3-D tori.

3.2 Complexity of CPE

Here we discuss the complexity of CPE. Assuming that an equal number of tasks are mapped to each processor, each processor holds $\nu = \left\lceil \frac{\nu_T}{\nu_P} \right\rceil$ tasks. Also, assuming that $\Delta(T)$ is $O(1)$, computing $best(r,q)$ and $best(q,r)$, for $q,r \in V_P$, costs 2ν. One sweep takes k steps, where $k = \chi'(P)$. At each step there are at most $\frac{\nu_P}{2}$ such pairs of processors computing *best*. Therefore, each sweep has of CPE has serial complexity

$$O\left(k\left(\frac{\nu_P}{2}\right)2\nu\right)$$

or

$$O\left(k\left(\frac{\nu_P}{2}\right)2\left\lceil\frac{\nu_T}{\nu_P}\right\rceil\right).$$

Simplifying this yields

$$O(k\nu_T).$$

However, k is a low order term that is either a constant or grows slowly as a function of P. For example, k is a constant for grids and tori and k grows as the $\log_2 \nu_P$ for hypercubes. Therefore, the serial complexity of CPE is

$$O(\nu_T).$$

Since the procedure *best* can be computed by all pairs of processors in parallel, the parallel complexity of CPE is

$$O\left(\frac{\nu_T}{\nu_P}\right).$$

3.3 Description of Test Cases

We use a variety of test cases to verify the operation of CPE. These are listed in Table 3.1. Test cases *3elt*, *4elt*, and *4elt-2* are triangular grids around multi-element airfoils; *bump* is a triangular grid over a bump; *motor* is the graph of a matrix from a nonlinear magnetostatic model of a permanent magnet motor, using an unstructured

Test Case	ν_T	ε_T	$\delta(T)$	$\Delta(T)$	dim.
3elt	4720	27444	3	9	2-D
4elt	15606	91756	3	10	2-D
4elt-2	11143	65636	3	12	2-D
bump	9800	57978	3	8	2-D
motor	6517	126306	7	44	2-D
bracket	62631	733118	3	32	3-D
rotor	99617	1324862	5	125	3-D
viking	156317	2118662	3	44	3-D

Table 3.1: Description of Test Cases.

finite element mesh with mixed triangular and quadrilateral third-order elements [6]; *bracket* is from structural mechanics [69]; *rotor* is an adapted tetrahedral grid around the helicopter rotor wing NACA 0015 [74]; *viking* is a tetrahedral grid describing the Lockheed S-3A Viking aircraft.

The table shows the number of vertices, the number of edges, the minimum degree and the maximum degree of the task graph for each test case. Also, the last column tells whether the problem is two dimensional or three dimensional. Pictures of most of the test cases as well as a histogram of the vertex degrees of all test cases are given in Appendix 1.

3.4 Calibration of Message Cycles and Communication Time

Before we verify the operation of CPE with the test cases, we first have to show that the communication time is a function of the number of message cycles. Then we show that reducing Λ_1 reduces the number of message cycles. CPE reduces Λ_1 which approximates the number of message cycles required to schedule the communications for a particular assignment of tasks to processors. In this section we calibrate the message cycles and actual communication time on the CM-2 for the one-to-many operation and then in the next section we verify the choice of Λ_1 as the objective function.

Recall from (1.5) that the time to send one or more 32-bit words on the CM-2 using version 6.0 of the communication compiler [20] is:

$$time_{CM} = (26 + 51 msg\_cycles + VPratio(17src + 57rcv))\ \mu sec.$$

The startup cost is 26 μsec. The quantity *msg_cycles* are the number of message cycles; *src* is the maximum number of unique 32-bit words being sent from any processor; *rcv* is the maximum number of unique 32-bit words being received by any processor. For the one-to-many operation, we can rewrite this as

$$time_{CM} = (51 msg\_cycles + 26 + \left\lceil \frac{\nu_T}{proc} \right\rceil [17 + 57(\Delta(T))])\ \mu sec, \quad \text{or}$$

$$time_{CM} = (51 msg\_cycles + \text{ overhead})\mu sec.$$

We use the number of 1-bit processors (*proc*) rather than ν_P above since this timing model is a function of the amount of work done by each 1-bit processor. Once we know *proc* and the task graph that we are mapping we can calculate the overhead. In a load-balanced mapping, the overhead is independent of the mapping.

In the next three figures we show the number of message cycles, the communication time predicted by (1.5) in msec, and the measured communication time msec the one-to-many communication on the CM-2. Each test was performed 1000 times and the average time is given. The different message cycles resulted from using different combinations of options to the communication compiler. We do not discuss the options since the communication compiler is not the focus here, it is used only as a tool to evaluate the results of CPE. The timings are obtained from CM Fortran.

Figure 3.3 shows different numbers of message cycles and times for the 3elt test case. Using information from Table 3.1, we can compute the overhead $= 26 + \left\lceil \frac{7420}{8192} \right\rceil (17 + 57 \cdot 9) = 556$ μsec (per iteration). The predicted time is consistently about 0.5 msec less than the actual time.

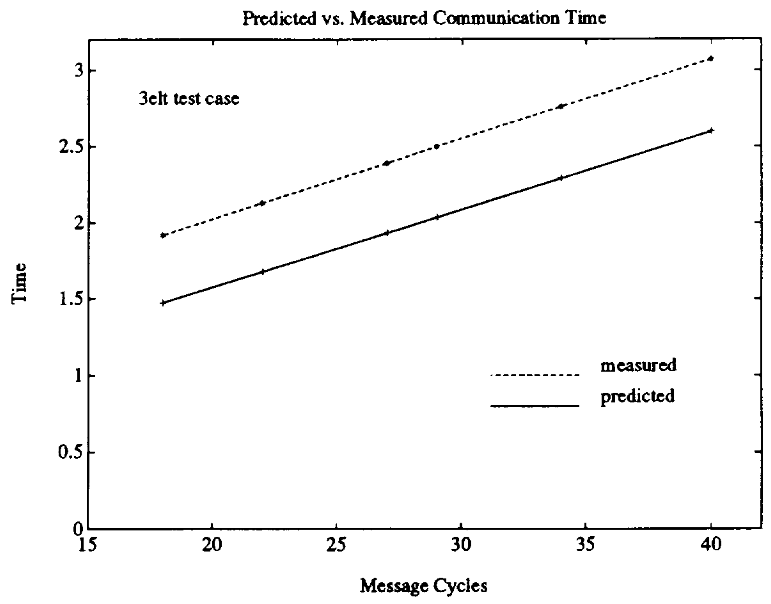


Figure 3.3: Calibration of Message Cycles and time in msec for the one-to-many operations on 3elt test case on 8K CM-2.

Figure 3.4 shows different numbers of message cycles and times for the 4elt test case. The overhead $= 26 + \left\lceil \frac{15606}{8192} \right\rceil (17 + 57 \cdot 10) = 1200$ μsec. The predicted time is approximately 0.9 msec less than the actual time in each case.

Figure 3.5 shows different numbers of message cycles and times for the bracket test case on a 16K CM-2. The overhead $= 26 + \left\lceil \frac{62631}{16384} \right\rceil (17 + 57 \cdot 32) = 7390$ μsec. For this test case, the predicted time is between 0.4 and 1.5 msec less than the actual time.

These tables show that the communication time is a function of the number of message cycles plus the overhead. In general, there is close correlation between the predicted communication time as a function of the number of message cycles and the actual communication time. When the number of message cycles is large, the 51 μsec per message cycle time dominates the communication time. Therefore, reducing the number of message cycles reduces the communication time.

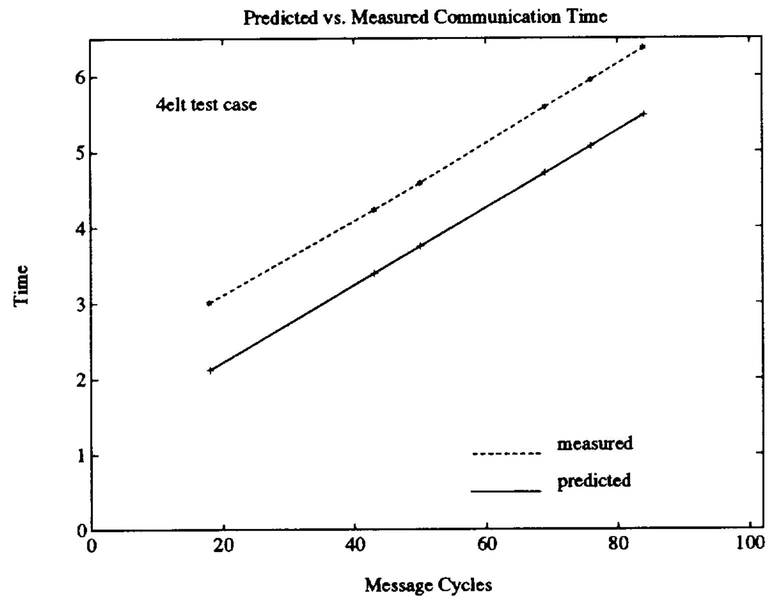


Figure 3.4: Calibration of Message Cycles and time in msec for the one-to-many operations on 4elt test case on 8K CM-2.

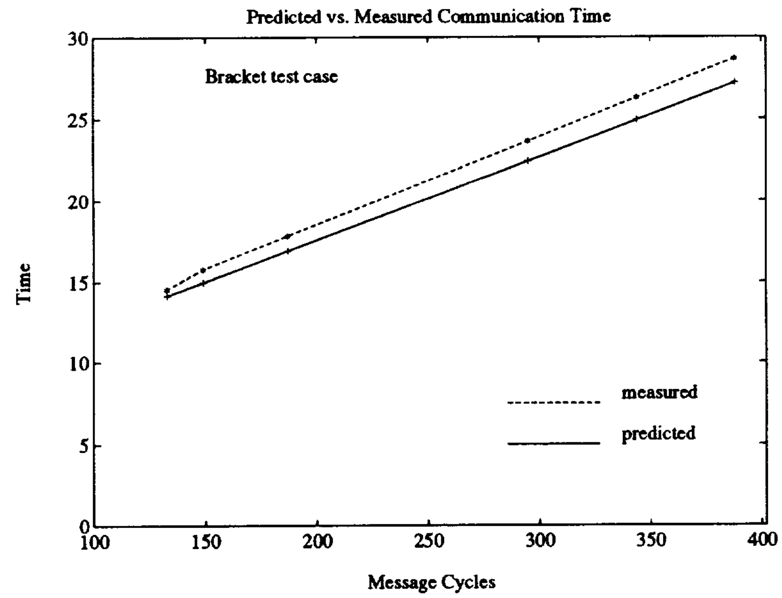


Figure 3.5: Calibration of Message Cycles and time in msec for the one-to-many operations on bracket test case on 8K CM-2.

	4elt, $\Delta(T) = 10$		motor, $\Delta(T) = 44$		bracket, $\Delta(T) = 32$	
	initial	mapped	initial	mapped	initial	mapped
sweeps	-	51	-	25	-	133
Λ_1	116442	48284	350238	115776	1447634	358688
$\bar{\Lambda}$	28.42	11.78	85.50	28.26	353.42	87.57
Λ_∞	8	8	8	6	8	7
msg_cycles	47	22	89	21	371	72
0	34704	47985	30314	47324	293208	422965
1	17280	20717	9682	45421	36080	179620
2	10390	5272	12759	19721	64370	55298
3	6169	1477	19111	5844	85065	10846
4	3927	513	20849	1260	97583	1601
5	2098	109	16193	193	66261	153
6	1061	49	8124	26	23606	3
7	393	23	2371		3834	1
8	128	5	386		480	

Table 3.2: Comparison of Λ_1 versus Λ_∞ for predicting number of message cycles, naive initial mapping.

3.5 Verification of Objective Function

We now show the correlation between the objective function Λ_1 and the number of message cycles. Also, we compare Λ_1 and Λ_∞ as predictors of the number of message cycles required to schedule a one-to-many communication for three test cases.

In Table 3.2 we compare Λ_1 and Λ_∞ after mapping the 4elt, motor, and bracket test cases with CPE. For each test case, we make comparisons for a naive initial mapping (defined in the next section) and after CPE has been used to map the task graph. We show the number of sweeps of CPE used, the values of Λ_1, $\bar{\Lambda}$, Λ_∞, message cycles, and a histogram of the distances that each word will travel for each mapping.

Recall that Λ_1 is the weighted sum of the distances that data must travel for a given mapping. One can think of it as the total communication load generated by

$\Psi(T)$. The bandwidth is the total communication capacity of P. The load divided by the capacity,

$$\bar{\Lambda} \equiv \frac{\Lambda_1}{bandwidth}, \tag{3.1}$$

should be a lower bound on the number of message cycles required. (As mentioned above, on an 8K CM-2 the bandwidth is 4096 words/cycle.) In general, the number of message cycles will be greater than $\bar{\Lambda}$. The number of message cycles is the true communication overhead and $\bar{\Lambda}$ is an approximation that assumes that all wires are equally loaded, which typically does not occur.

On the other hand, the scheduling of data to wires can be optimized for the one-to-many operation, if one word is being sent to several tasks mapped to the same processor. In this case, the word can be sent once and replicated at the destination rather than being sent once for each receiving task. This explains why the number of message cycles is less than $\bar{\Lambda}$ for the motor and bracket test cases.

Table 3.2 shows that for the three test cases, CPE reduced $\bar{\Lambda}$ by a factor of 2.41, 3.0, and 4.03 and the number of message cycles was reduced by factors 2.1, 4.2, and 5.1, respectively. On the other hand, Λ_∞ was reduced from 8 to 6 for the motor test case, from 8 to 7 for the bracket test case and not at all for the 4elt test case. The histogram of distances shows that it is more important to reduce the distances that each piece of data must travel rather than the maximum distance that any piece of data must travel. We see that $\bar{\Lambda}$ is a good predictor of the number of message cycles on the CM-2, Λ_∞ is not.

3.6 Initial Mappings – Definition and Comparison

As mentioned above, CPE starts with some initial assignment of tasks to processors. Here we define and compare four different initial mappings used in conjunction with CPE: naive, random, gray code, and RSB. RSB and gray code initial mappings were defined in Chapter 2, but are defined again here.

The first initial mapping is a *naive* mapping. When discretizing grids are created the grid points are usually given some numbering such as the order that they are generated – the first grid point generated would be labeled number 1, the second labeled number 2, etc. Let there be one task associated with each grid point, task 1 associated with grid point 1, ..., task i associated with grid point i, etc. A *naive* embedding of the task graph to the processor graph assigns task 1 to processor graph vertex 1, ..., task i to processor graph vertex i, and so forth. If $m \equiv \left\lceil \frac{\nu_T}{\nu_P} \right\rceil$, then for a naive mapping, tasks 1 to m are mapped to processor 1, tasks $(m+1)$ to $2m$ are mapped to processor 2, etc.

The second initial mapping we use is a *random* mapping scheme, a random permutation of the task graph vertex numbers followed by a naive initial mapping of the permuted values.

The third initial mapping is a *gray code* mapping. Recall that in a gray code initial mapping, one first computes the binary gray code of each task number and then computes a naive mapping of the binary gray code values of each task.

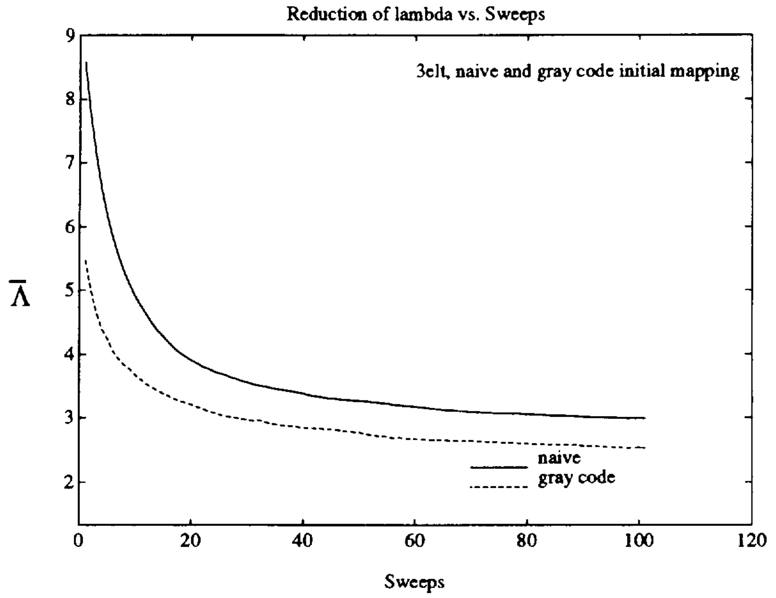
The fourth initial mapping used is RSB. Pothen, Simon, and Liou [55] and Simon [70] they showed that RSB produced lower cost partitions than ROB or nested dissection partitioning techniques. RSB is the only partitioning scheme from Section 2.4.1 that we use as an initial mapping. RSB initial mapping is a two step process. First, we use RSB to compute a ν_P-way partition of the task graph. Then, we use CPE to embed the associated quotient graph in P, performing pairwise exchanges on groups of task graph vertices represented by the quotient graph vertices. The embedding starts with a naive assignment of vertices from the quotient graph to the processors.

3.6.1 Comparison of Initial Mappings

We have developed two implementations of CPE. One is a parallel implementation written in *lisp that runs on the CM-2 and the other is a serial implementation written in *C*. The parallel code maps task graphs to the CM-2 and is machine specific and differs slightly from the serial implementation. On the CM-2, the number of tasks mapped to any sprint node is always an integer multiple of 32, since there are 32 1-bit processors in each sprint node. To accomplish this, we augment the task graph with *wild card vertices.* A *wild card vertex* is a vertex with no neighbors. For example, in the 3elt test case $\varepsilon_T = 4720$ so we add $8192 - 4720 = 3472$ wild card vertices to T. In the parallel implementation of CPE, up to 32 tasks for the 3elt test case will be mapped to any sprint node. The serial implementation maps no more than m tasks to each processor, so using CPE to map T to an 8-cube results in some processors having 19 tasks mapped to them and some having 18 tasks mapped to them. Also, in the serial implementation, the user can specify the dimensions and type of processor graph. The current options are hypercube, 2-D or 3-D grid, and 2-D or 3-D torus. Some of the experiments here are done using the serial code and some are done using the parallel code.

Figures 3.6 through 3.11 compare the results of using the parallel implementation of CPE in conjunction with naive, gray code, and random initial mappings for the 3elt, 4elt, and motor test cases. In each figure there are two graphs. The upper one shows the reduction of $\bar{\Lambda}$ and the lower one shows the reduction of the number of message cycles as a function of the number of CPE sweeps.

For the data in Figures 3.6, 3.8, and 3.10, two different initial mappings are used. The first mapping is a naive mapping denoted by the solid line and the second is the gray code initial mapping denoted by the dashed line. In Figures 3.7, 3.9, and 3.11, three different random initial mappings are used followed by CPE. The number to the right of the lower graph in each figure is the smallest number of



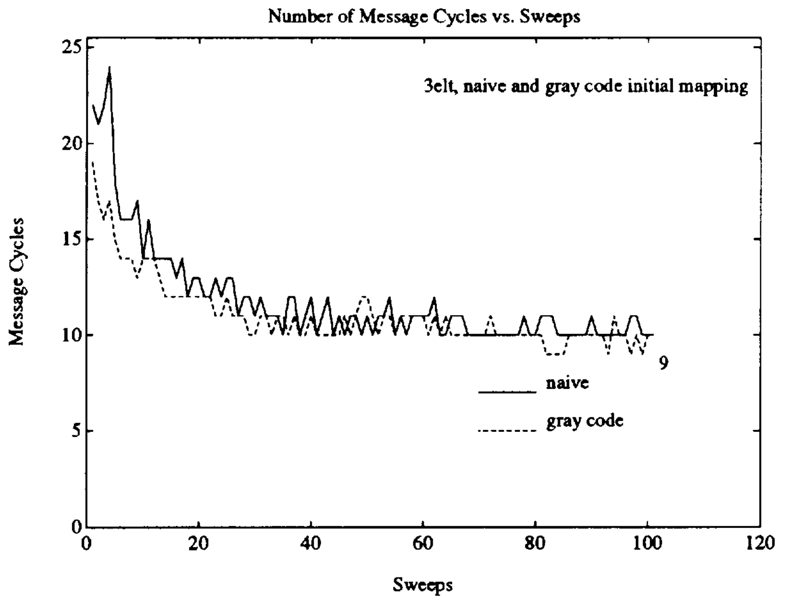
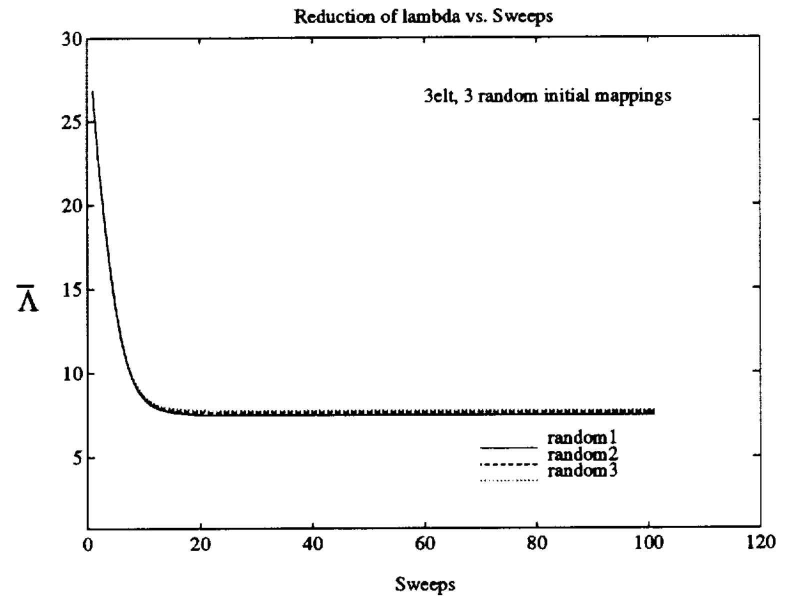


Figure 3.6: Reduction of $\bar{\Lambda}$ and message cycles versus sweeps after naive and gray code initial mappings on 3elt task graph.



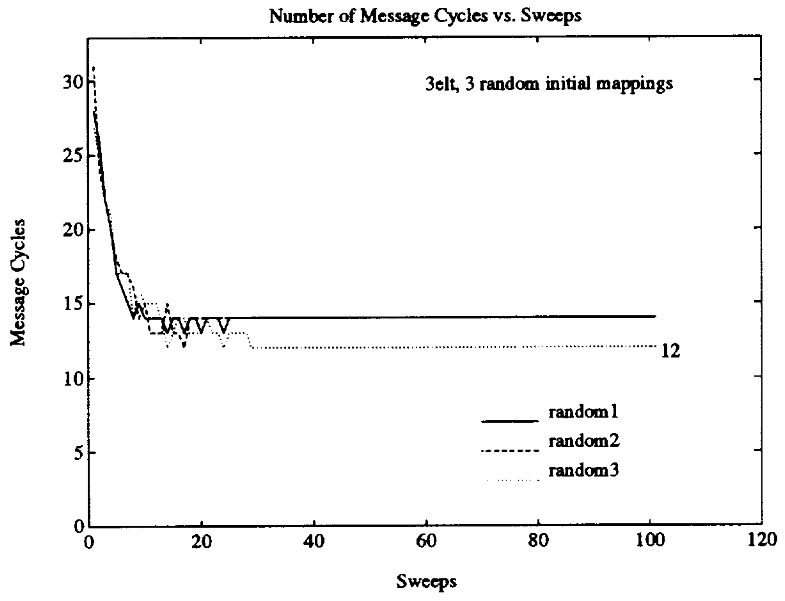
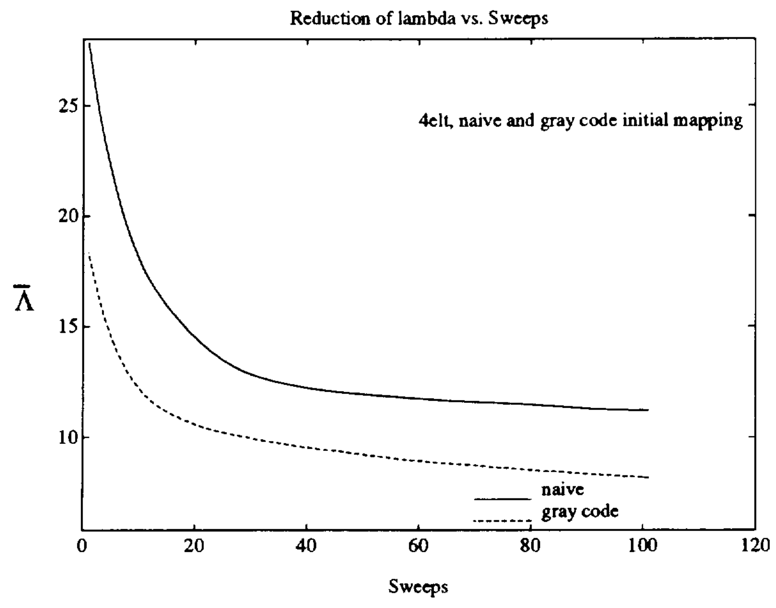


Figure 3.7: Reduction of $\bar{\Lambda}$ and message cycles versus sweeps with 3 random initial mappings on 3elt task graph.



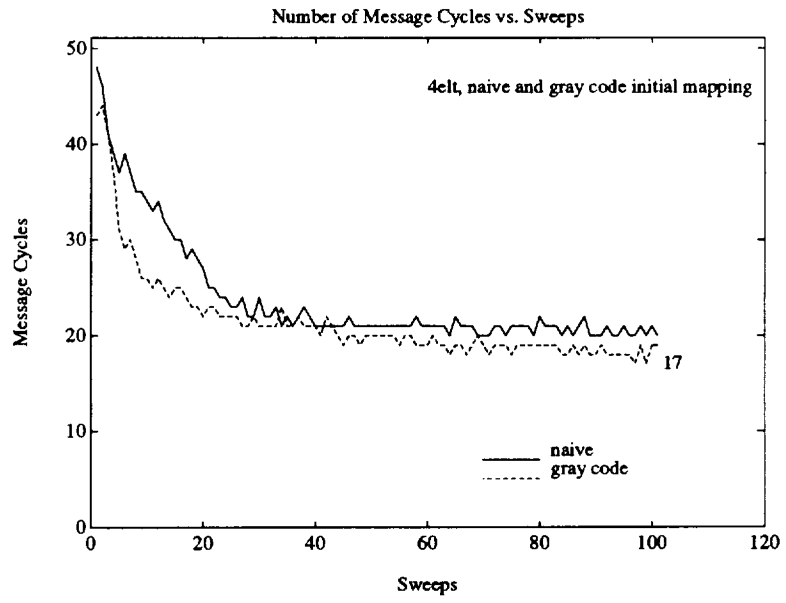
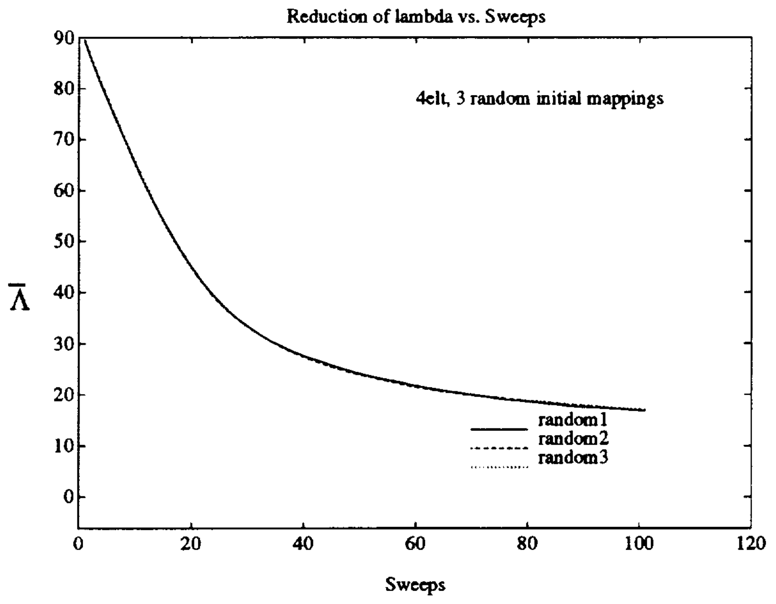


Figure 3.8: Reduction in $\bar{\Lambda}$ and message cycles versus sweeps using naive and gray code initial mappings on 4elt task graph.



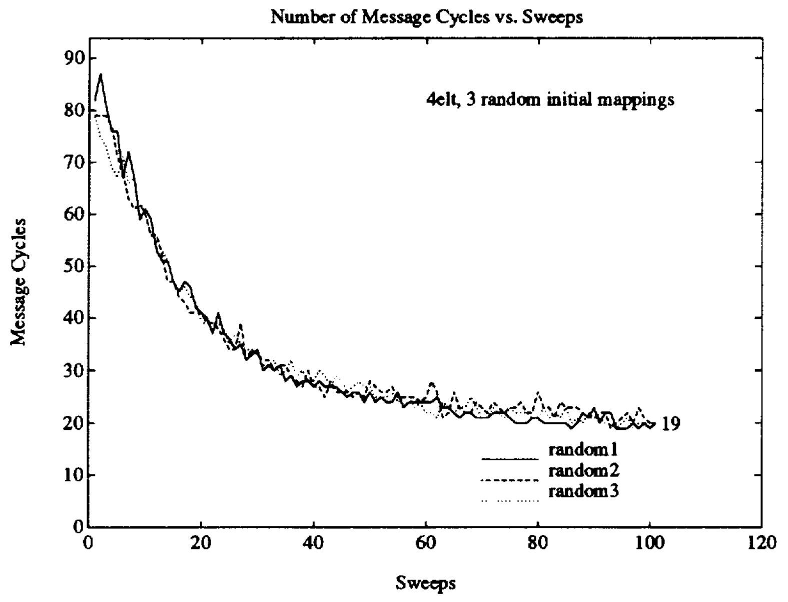
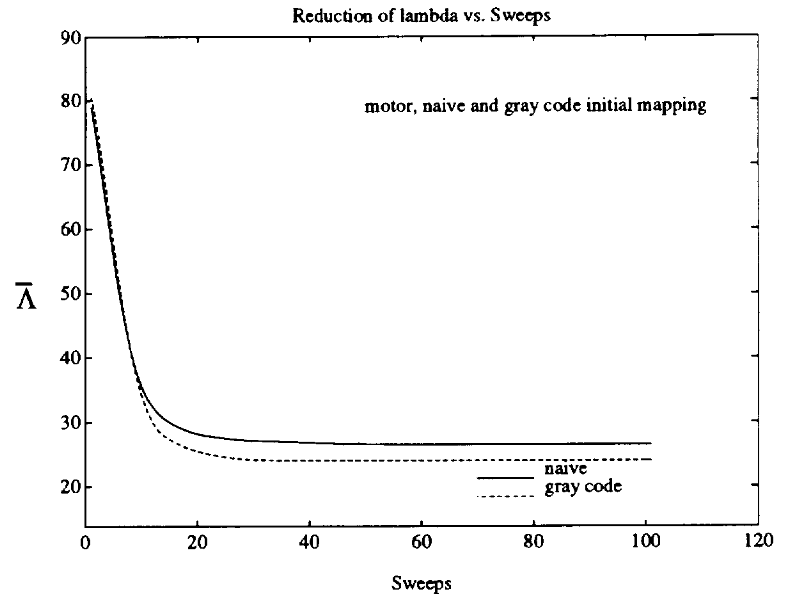


Figure 3.9: Reduction in $\bar{\Lambda}$ and message cycles versus sweeps using 3 random initial mappings for 4elt task graph.



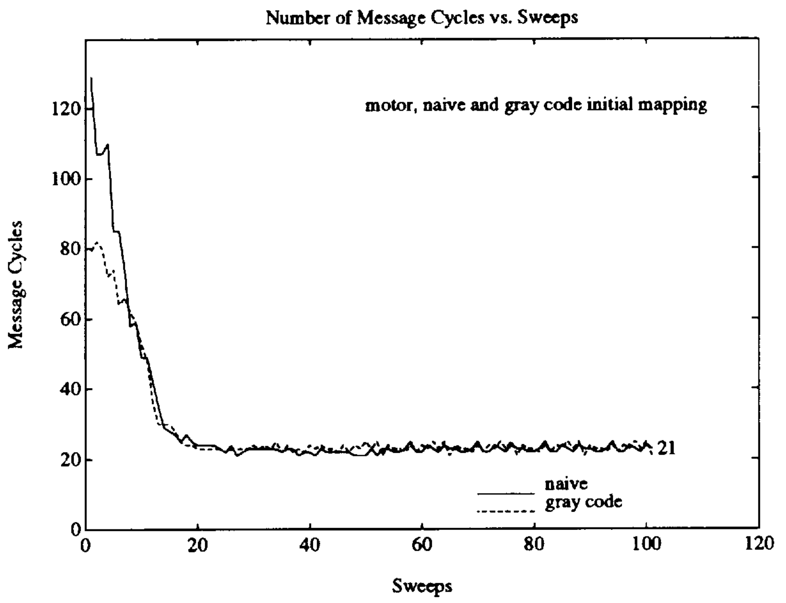
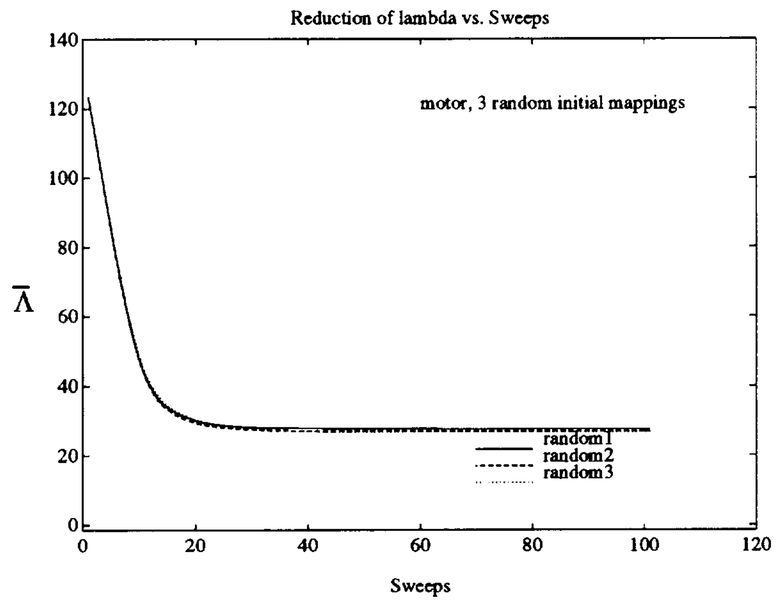


Figure 3.10: Reduction of $\bar{\Lambda}$ and message cycles versus sweeps after naive and gray code initial mappings on motor task graph.



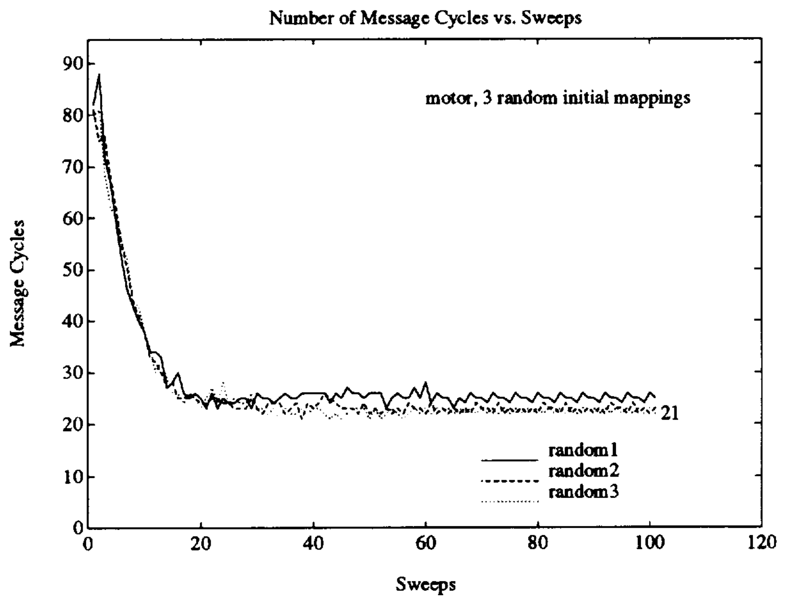


Figure 3.11: Reduction of $\bar{\Lambda}$ and message cycles versus sweeps with 3 random initial mappings on motor task graph.

message cycles achieved in this figure.

These graphs illustrate several points. The heuristic effectively reduces $\bar{\Lambda}$ and the number of message cycles. Qualitatively, the upper and lower graphs in each figure have the same "shape" which indicates that $\bar{\Lambda}$ is a good predictor of the number of message cycles and thus the communication time on the CM-2. Also, CPE is sensitive to the initial mapping. In these three test cases, the naive and gray code initial mappings resulted lower numbers of message cycles than any of the three random initial mappings.

We cannot explain the oscillations in the lower graphs showing number of messages cycles versus the CPE sweeps. We conjecture that the communication compiler is very sensitive to small perturbations in the mapping.

In Figures 3.12 through 3.19 we use the serial implementation of CPE to compare initial mappings for all test cases. We compare the mapping achieved by a naive initial mapping followed by CPE, a random initial mapping followed by CPE, and two different RSB initial mappings followed by CPE. The first RSB initial mapping starts with a random embedding of the quotient graph in the processor graph and the second starts with a naive embedding of the quotient graph in the processor graph. Both embeddings are improved using CPE. Finally, the partitioning information is ignored and CPE is used on individual tasks. In each trial, we map the task graph to an 8-cube, mapping an equal number of tasks to each processor.

In the curves corresponding to the RSB initial mapping, the point where the line appears to level out is where the switch from pairwise exchange of partitions to pairwise exchange between tasks occurs. We have duplicated the value of $\bar{\Lambda}$ to mark the change.

In Figures 3.17 through 3.19 we show two graphs each. The top one depicts all four sets of data graphed together and the bottom one shows just the two RSB curves.

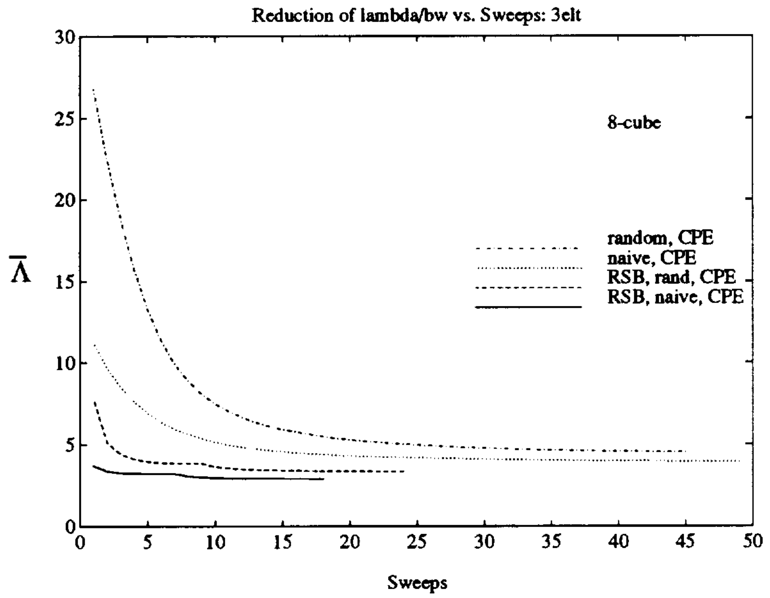


Figure 3.12: Reduction of $\bar{\Lambda}$ versus sweeps for 3elt test case.

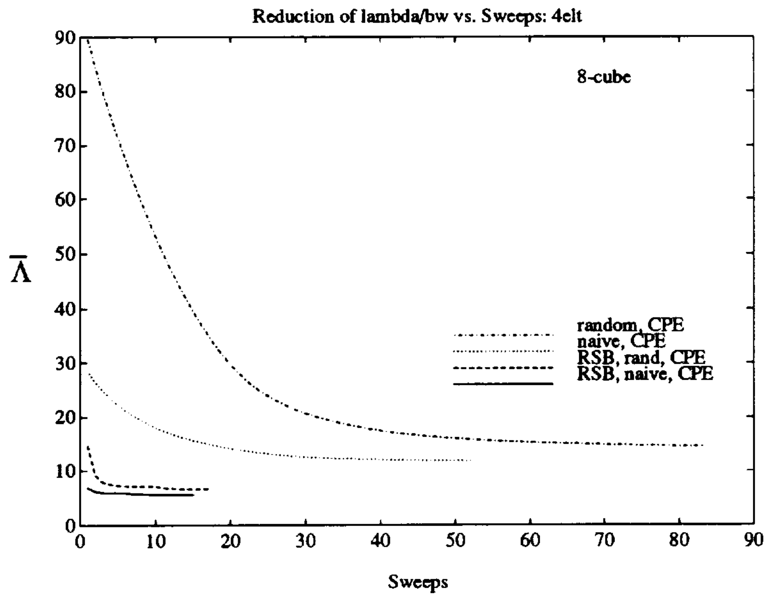


Figure 3.13: Reduction of $\bar{\Lambda}$ versus sweeps for 4elt test case.

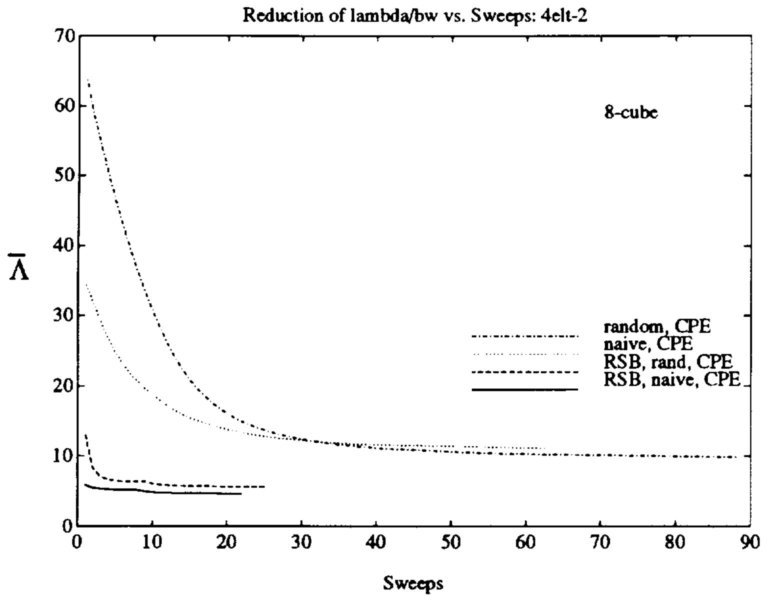


Figure 3.14: Reduction of $\bar{\Lambda}$ versus sweeps for 4elt-2 test case.

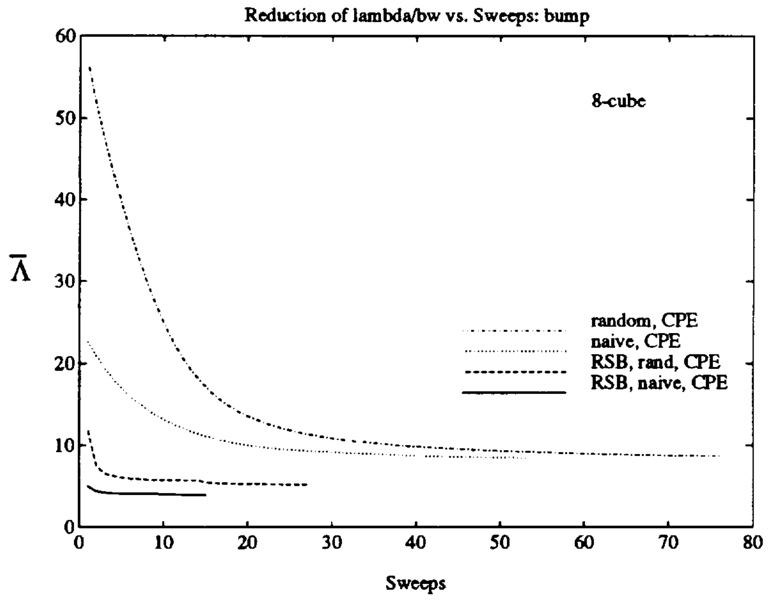


Figure 3.15: Reduction of $\bar{\Lambda}$ versus sweeps for bump test case.

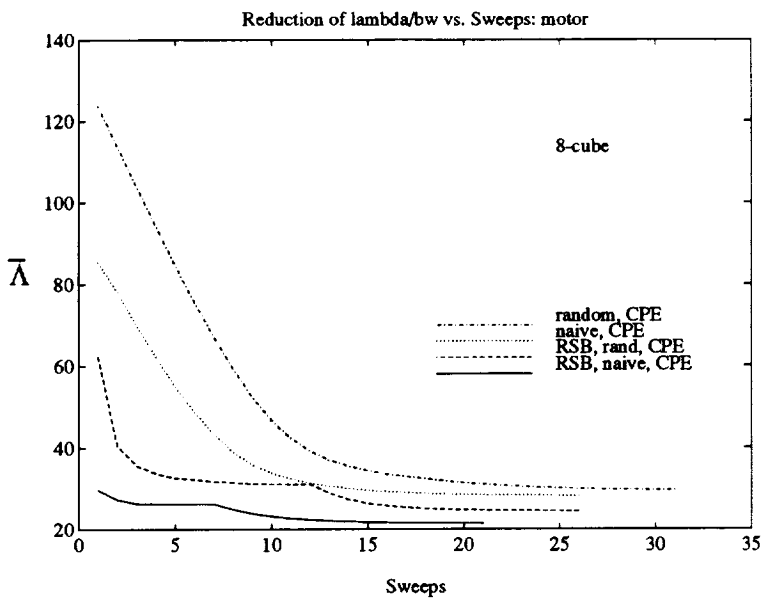
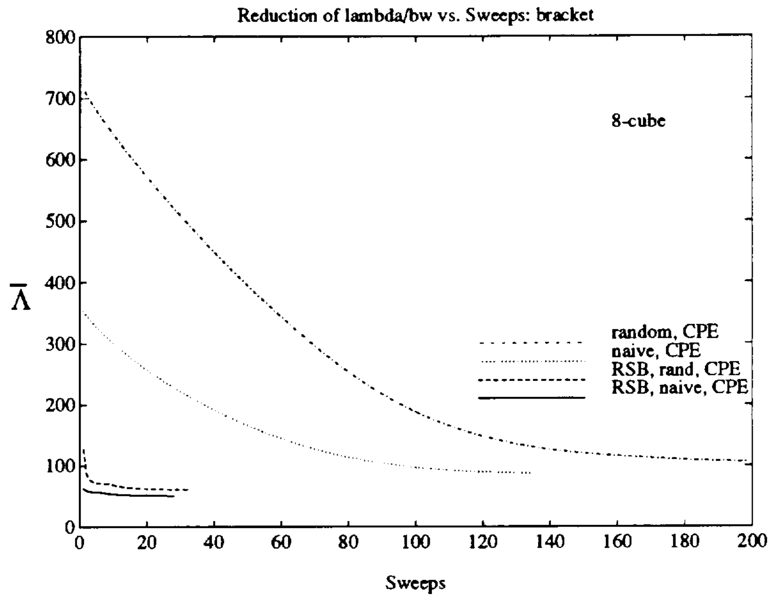


Figure 3.16: Reduction of $\bar{\Lambda}$ versus sweeps for motor test case.

These figures show that an RSB initial mapping followed by CPE achieves better mappings than random, naive, or gray code initial mappings followed by CPE.

3.7 Comparison of Local, Distance 2 and Global Searches

As mentioned above, in each sweep, a task $v \in V_T$ has the opportunity to exchange its processor assignment with any task $u \in V_T$, if $d(\Psi(v), \Psi(u)) = 1$. But, one might wonder whether better mappings are possible if a task u could exchange places with a larger subset of V_T. In this section, we quantify the difference in mappings if the size of the subset is increased. We compare the serial implementation of CPE as described above with two variations. In the first variation, a task $v \in V_T$ has the opportunity to exchange its processor assignment with any task $u \in V_T$, if $d(\Psi(v), \Psi(u)) = 2$. We call this *distance-2* CPE since a task v can be exchanged with a task mapped to a processor at distance one or distance two from $\Psi(v)$. In the implementation of this algorithm, we find a proper coloring of all paths of length one and two and use CPE as before. In the second variation, a task v exchanges



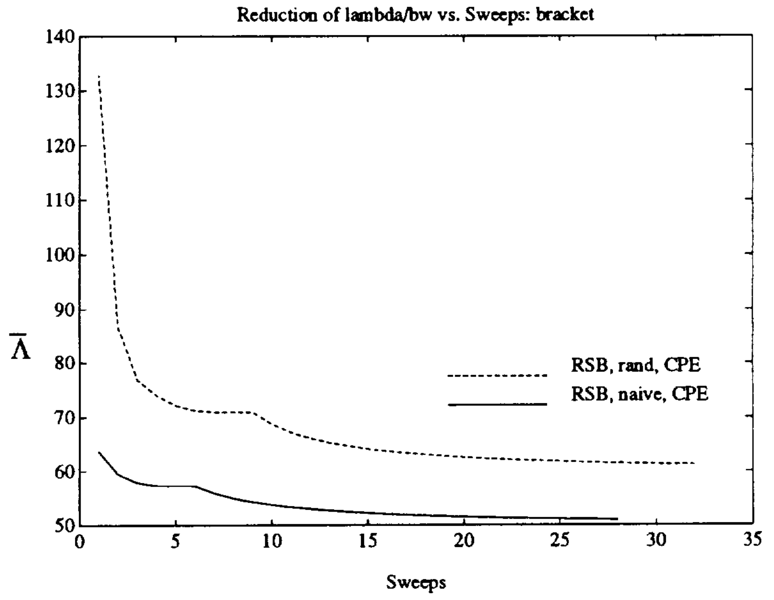
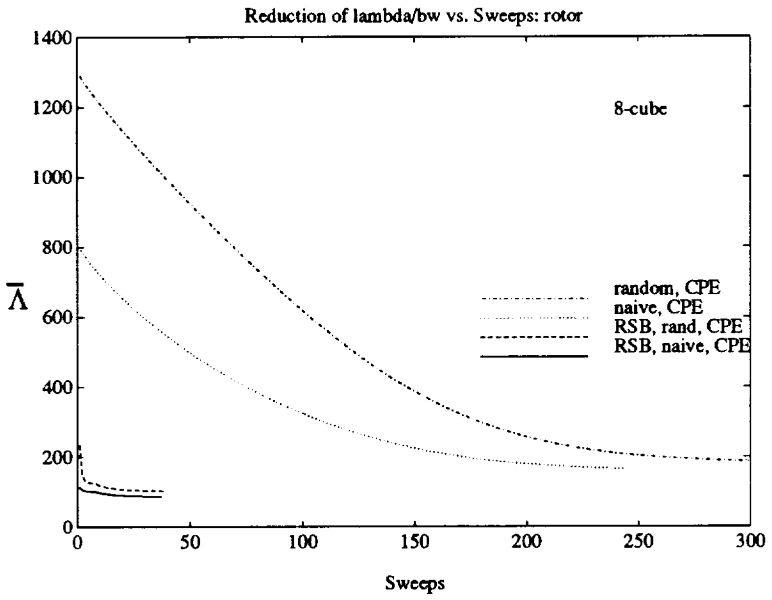


Figure 3.17: Reduction of $\bar{\Lambda}$ versus sweeps for bracket test case.



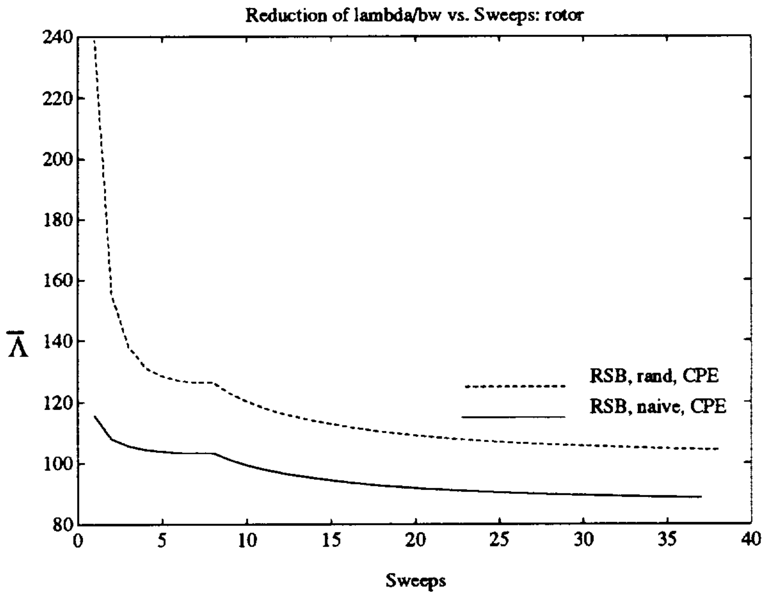
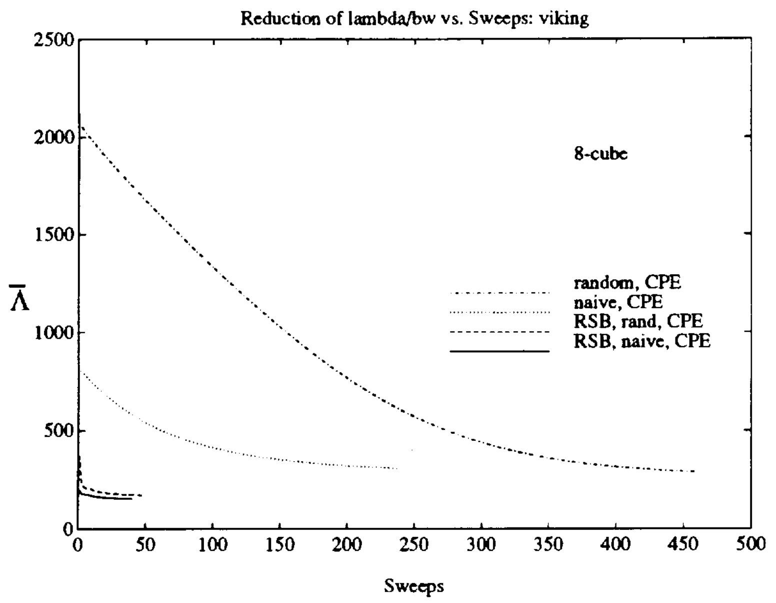


Figure 3.18: Reduction of $\bar{\Lambda}$ versus sweeps for rotor test case.



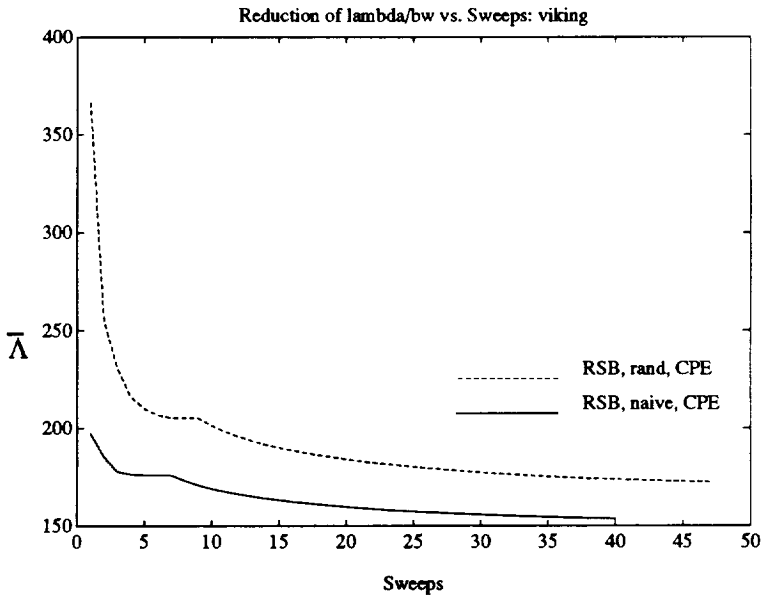


Figure 3.19: Reduction of $\bar{\Lambda}$ versus sweeps for viking test case.

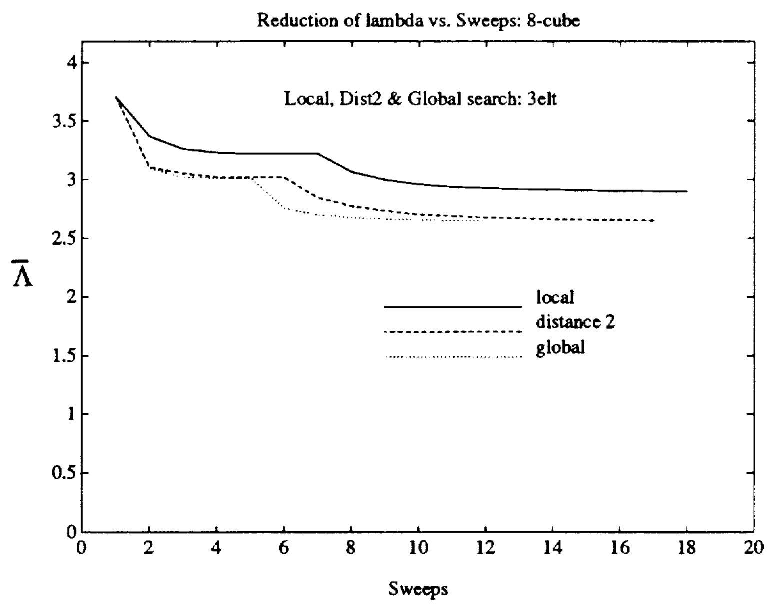


Figure 3.20: Reduction of $\bar{\Lambda}$ versus sweeps for three search distances on 3elt test case.

its mapping with any task u such that $\Psi(v) \neq \Psi(u)$. We call this *global* CPE, since a task can have its location exchanged with any other task. In the implementation of global CPE, we exchange the mapping of task v with the task that causes the largest reduction in Λ_1. Note that global CPE is a sequential $O(\nu_T^2)$ algorithm.

In Figures 3.20 and 3.21 we compare the local CPE with distance-2 and global CPE. In Figure 3.22 we compare local CPE with distance-2 CPE. We map the 3elt, 4elt, and bracket test cases to an 8-cube. In each figure, the distance-2 search results in a better mapping than the local search CPE, but only by a small percentage. The running time of the distance-2 search is 4.5 times greater than the local search since there are 36 distance one and two neighbors from each processor and only 8 distance one neighbors in an 8-cube. In general, the number of distance one and two neighbors for an n-cube is $(n(n+1))/2$. As n increases, the complexity of distance-2 CPE on an n-cube increases like $O(n^2)$ and the complexity of local CPE is increasing $O(\log n)$.

In the first two examples, the global search is better than the other two, but

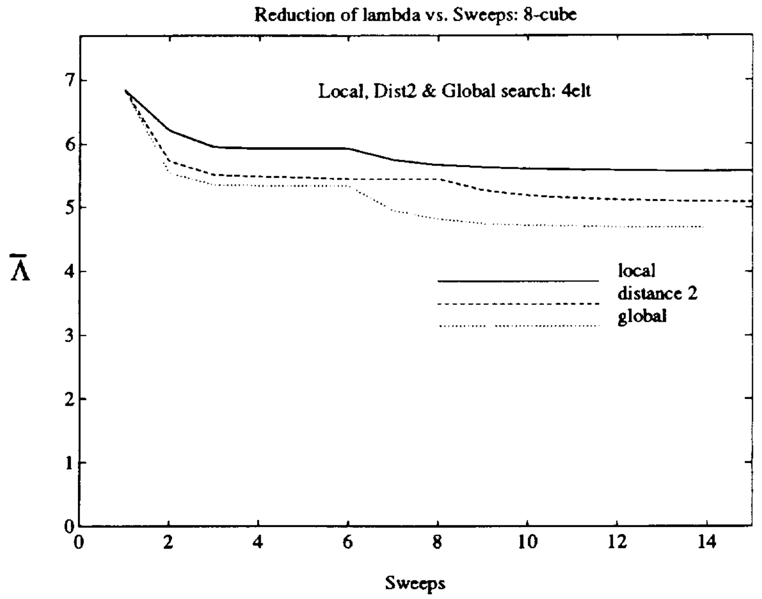


Figure 3.21: Reduction of $\bar{\Lambda}$ versus sweeps for three search distances on 4elt test case.

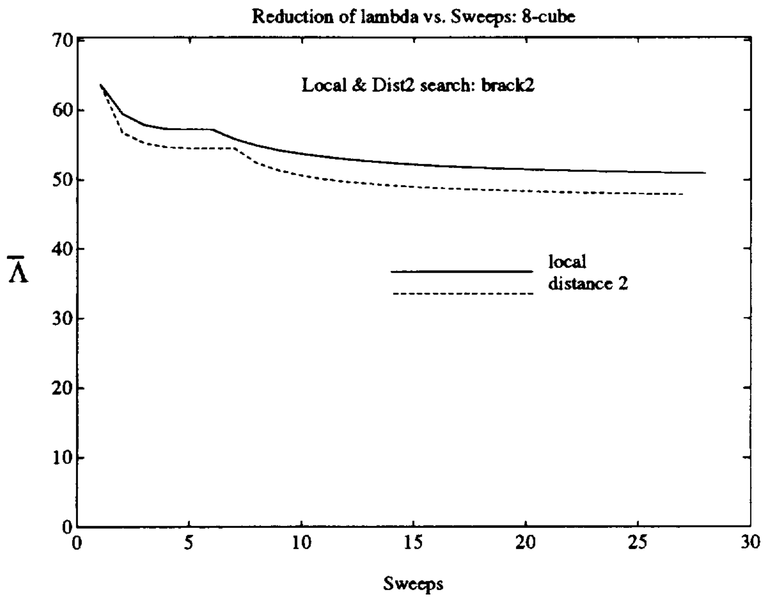


Figure 3.22: Reduction of $\bar{\Lambda}$ versus sweeps for two search distances on bracket test case.

again, by a small percentage. The global search is $O(\nu_T^2)$. We did not compute the mapping for the bracket test case, since the global search on the 3elt test case took over 2 hours. The bracket test case has 13.27 times more vertices so it would take approximately 352 cpu hours (14.6 days) on a Sun-4 workstation.

These graphs show that when mapping these three test cases to an 8-cube, the mappings produced by CPE result in only slightly larger values of $\bar{\Lambda}$ than variants that examine a larger subset of V_T, at a significant time savings.

3.8 Comparison with Simulated Annealing

Here we compare CPE with the parallel SA heuristic mapping algorithm developed by Dahl [19]. In Tables 3.3, 3.4, and 3.5 we show the number of message cycles that results from using the SA mapping on the 3elt, 4elt, and motor test cases. Each table shows 200 experiments on one of the test cases, varying the beginning temperature and the number of iterations. The ending temperatures used to compute the tables were fixed at 0.1, 0.05, and 0.05, respectively. We did not make an exhaustive search through the possible values of the three parameters but ran enough examples to gain confidence that the values shown are representative of the mappings possible when using SA.

In Table 3.6 we compare the best mapping that we observed using SA, the best mapping we observed using parallel CPE and the best we obtained with RSB and the serial version of CPE. We show the number of message cycles required to schedule the one-to-many operation using the communication compiler and also the time required for the heuristics to achieve this particular mapping. We see that SA achieved a lower number of message cycles than the parallel CPE for one test case. For the motor test case, the mapping from parallel CPE required about 11.7% more message cycles; SA took 6.22 times longer to run.

One should note that the time for SA is for running exactly one test case. It

	Beginning Temperature									
# iter	0.8	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0
50	25	20	25	21	20	20	21	21	22	19
100	16	16	17	17	17	19	18	18	18	19
150	16	16	14	15	16	16	17	17	15	15
200	13	14	14	14	14	15	15	14	15	15
250	14	13	14	14	14	14	13	15	14	15
300	13	12	12	12	12	14	12	14	12	14
350	14	12	12	12	11	14	12	13	14	13
400	13	11	11	12	12	11	13	13	12	13
450	11	11	11	13	11	11	12	13	12	12
500	12	10	11	12	12	11	11	12	12	13
550	11	11	11	12	12	11	11	11	10	13
600	10	10	12	10	10	12	12	11	12	11
650	10	11	10	10	13	11	11	10	11	12
700	11	11	11	11	12	10	12	11	10	11
750	12	10	11	9	11	11	9	11	11	11
800	10	9	10	10	11	10	10	11	10	12
850	12	8	10	9	9	10	10	10	10	11
900	9	10	10	9	10	10	9	10	10	11
950	11	11	10	10	11	10	12	10	11	10
1000	11	10	10	11	10	10	11	11	10	11

Table 3.3: Number of Message Cycles as a function of beginning temperature and number of iterations using SA on the 3elt test case.

	Beginning Temperature									
# iter	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
200	41	42	42	41	42	43	43	45	43	45
400	32	34	33	33	37	36	33	38	35	37
600	29	32	31	31	31	30	32	34	34	34
800	27	25	29	28	28	26	30	28	29	28
1000	24	25	26	26	25	25	28	27	26	26
1200	23	27	25	25	27	26	27	24	24	24
1400	23	23	23	23	25	23	25	24	24	25
1600	22	23	21	25	23	23	24	25	23	27
1800	22	21	23	22	22	24	22	23	25	24
2000	21	23	20	22	21	22	22	22	22	23
2200	21	20	20	19	22	20	21	22	22	22
2400	20	20	21	20	21	21	25	22	20	22
2600	20	19	19	19	20	20	19	20	20	22
2800	18	20	19	20	21	20	21	20	23	21
3000	18	21	18	20	18	20	20	21	20	21
3200	18	21	18	**17**	20	18	20	19	21	20
3400	19	20	18	19	19	18	19	19	20	19
3600	18	19	18	19	20	18	20	20	19	19
3800	17	18	18	18	20	18	19	19	21	18
4000	21	18	18	18	19	17	18	19	22	20

Table 3.4: Number of Message Cycles as a function of beginning temperature and number of iterations using SA on the 4elt test case.

# iter	Beginning Temperature 1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
200	41	41	41	41	45	47	41	42	46	43
400	33	30	29	31	27	34	30	35	31	30
600	25	27	29	25	25	27	26	32	25	28
800	25	23	26	25	24	29	22	26	23	22
1000	24	21	24	25	23	24	26	25	27	27
1200	26	25	22	24	21	23	22	23	28	22
1400	24	22	25	22	20	24	21	24	22	20
1600	23	23	23	21	21	23	23	19	21	25
1800	22	23	22	24	22	21	21	21	23	22
2000	21	23	23	22	21	21	21	22	22	23
2200	26	23	20	19	19	21	19	22	21	19
2400	21	22	22	21	21	19	21	19	20	20
2600	22	22	20	20	24	21	23	19	21	19
2800	22	20	21	19	19	18	19	21	23	23
3000	20	24	20	22	19	18	21	19	19	21
3200	20	20	22	20	21	18	20	20	19	19
3400	25	21	24	18	20	19	18	19	22	21
3600	20	21	21	19	19	<u>**17**</u>	20	19	19	18
3800	24	20	21	27	20	24	18	18	18	29
4000	23	22	21	18	21	19	19	18	17	18

Table 3.5: Number of Message Cycles as a function of beginning temperature and number of iterations using SA on the motor test case.

test case	SA		Naive & Parallel CPE			RSB & Serial CPE	
	cycles	time	cycles	time	sweeps	cycles	sweeps
3elt	8	67	10	53	85	7	16
4elt	17	382	16	94	79	12	12
motor	17	840	20	135	55	19	19

Table 3.6: Comparison of SA and CPE on 3 test cases: best mappings and time in seconds for mapping.

took many hours to compute the table to determine the optimal parameter values. Ignoring the time to read the file from the front end, the running time for 1000 iterations for one set of temperature parameters of SA for the 3elt test case, the 4elt test case, and the motor test case is 40 seconds, 113.75 seconds, and 225 seconds, respectively. On an 8K CM-2, 1.16 hours, 13.27 hours, and 26.25 hours were required to compute the values in the tables for the three test cases. The time for the serial RSB and CPE are not included since it was run on a workstation and it would not be a meaningful comparison. The running time for a grey code initial guess followed by CPE is similar to the time for a naive initial guess followed by CPE, the grey codes for each task take a fraction of a second to compute in parallel.

We also measured the elapsed time for 100 sweeps of the parallel CPE on these three different test cases on 8K processors of the CM-2 hosted by a Sun-4. For test cases 3elt, 4elt, and motor, the time was 62.32 seconds, 118.54 seconds and 247.70 seconds, respectively.

These comparisons show that parallel CPE and SA are capable of achieving similar quality mappings. However, one must find a good combination of the three input parameters, otherwise SA yields mappings that are much worse than we achieve using and initial mapping and CPE. Using an RSB initial mapping followed by CPE does not require the user to choose initial parameters, the mappings are comparable to the best the SA can achieve. An open question is whether CPE can be followed by SA to some good effect.

3.9 Interconnection Comparison

Finally, we use CPE to study the communication capabilities of various interconnection schemes for a set of hypothetical, but plausible, parallel computers. We study three parallel computers: 256, 512, and 1024 processors and bandwidths 4096, 8192, and 16384 words/cycle, respectively. For each system, we analyze five different interconnection schemes: hypercube, 2-D grid, 2-D torus, 3-D grid, and 3-D torus. For example, for the 256-processor system we have an 8-cube, a 16×16 grid of processors (2D Grid), a 16×16 torus (2D Torus), a $8 \times 8 \times 4$ grid (3D Grid) and a $8 \times 8 \times 4$ torus (3D torus). Also, we normalize the bandwidths for each system size so that there is a width-two connection between processors in the hypercube and the interconnections in the other systems are proportionally wider. For example, for the 256-processor system, each processor in the 8-cube has 8 neighbors and each interconnection has width two. In the 16×16 torus, each processor has half as many neighbors so the connections between them are twice as wide, or four. In other words, for a fixed number of processors, we choose the width of each wire such that the machine bandwidth is constant in all systems, independent of the interconnection scheme.

In Figures 3.23 through 3.28 we show the many-to-many communication for the bracket and viking test cases on five configurations of three different machine sizes. We use an RSB initial mapping followed by distance-2 CPE rather than local CPE because we want the best mapping possible in a reasonable amount of time. (Recall that we showed that there was only a small difference between distance-2 CPE and global CPE on a hypercube.) This reduces the bias present in CPE against low-dimensional networks. The bias is due to the fact that CPE exchanges task between neighbors and that low-dimensional networks have a smaller number of neighbors than higher-dimensional networks, thus limiting the size of the subset of possible exchanges. Each figure shows the reduction of $\bar{\Lambda}$ as a function of the number of sweeps.

We cannot compare the different systems unless we know the ratio of $\bar{\Lambda}$ to the number of message cycles. Define Θ as

$$\Theta \equiv \frac{\bar{\Lambda}}{msg\_cycles}.$$

One can think of Θ as measuring the percent wire utilization.

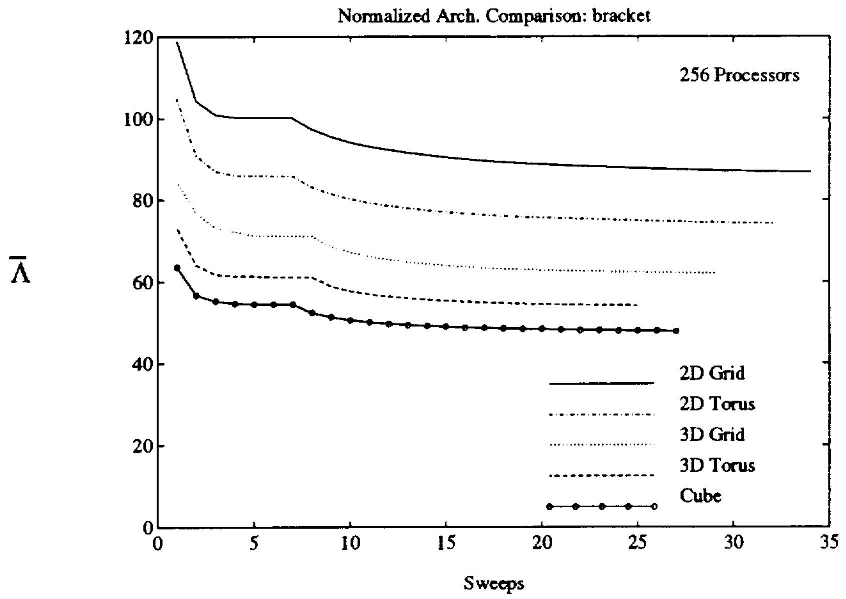


Figure 3.23: Comparison of communication capabilities for 5 interconnection schemes using the bracket test case mapped to 256 processors.

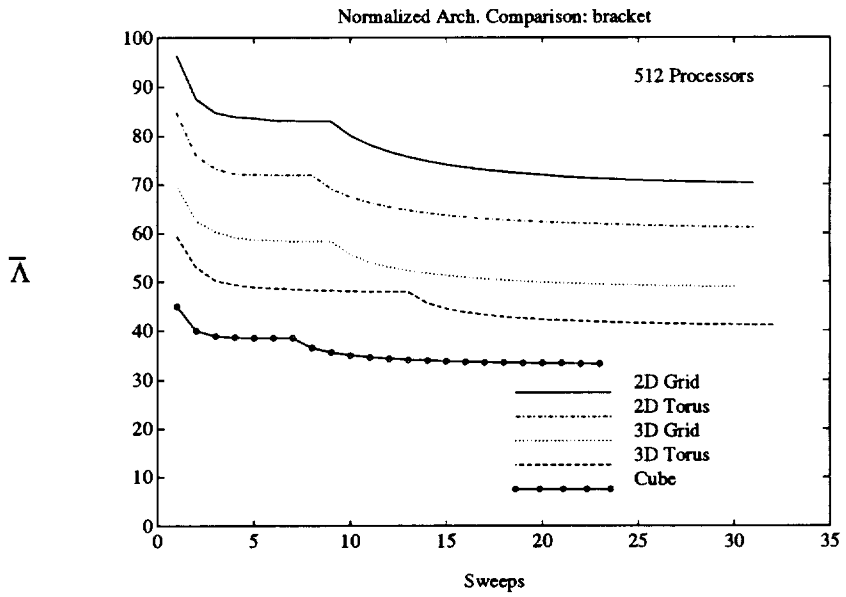


Figure 3.24: Comparison of communication capabilities for 5 interconnection schemes using the bracket test case mapped to 512 processors.

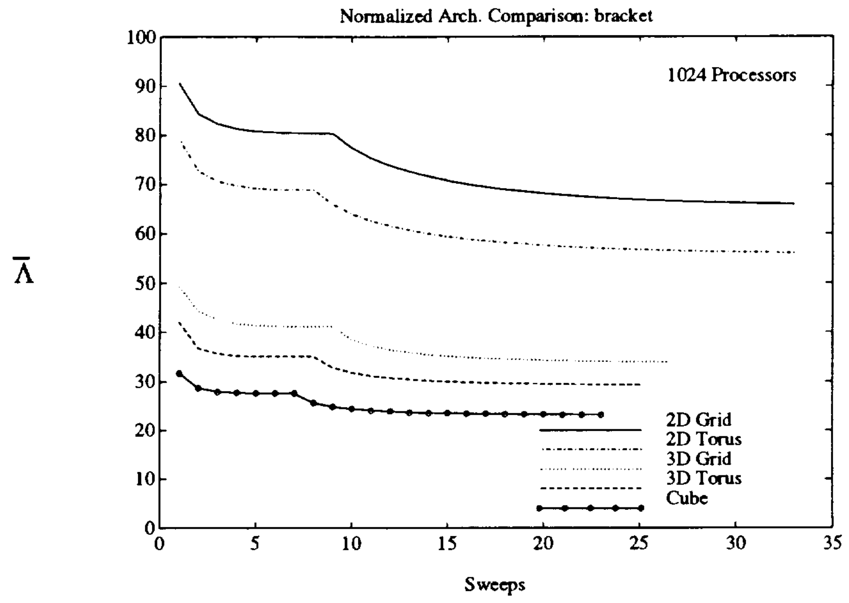


Figure 3.25: Comparison of communication capabilities for 5 interconnection schemes using the bracket test case mapped to 1024 processors.

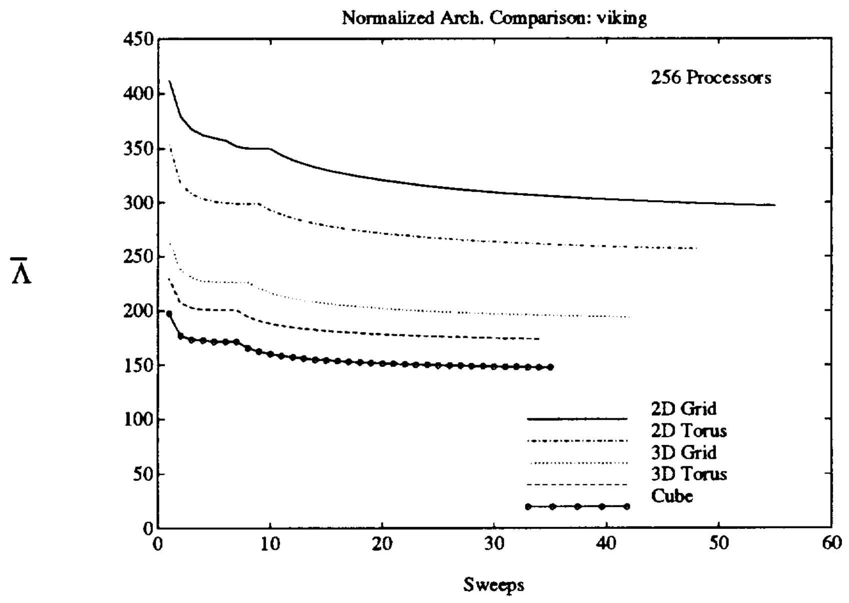


Figure 3.26: Comparison of communication capabilities for 5 interconnection schemes using the viking test case mapped to 256 processors.

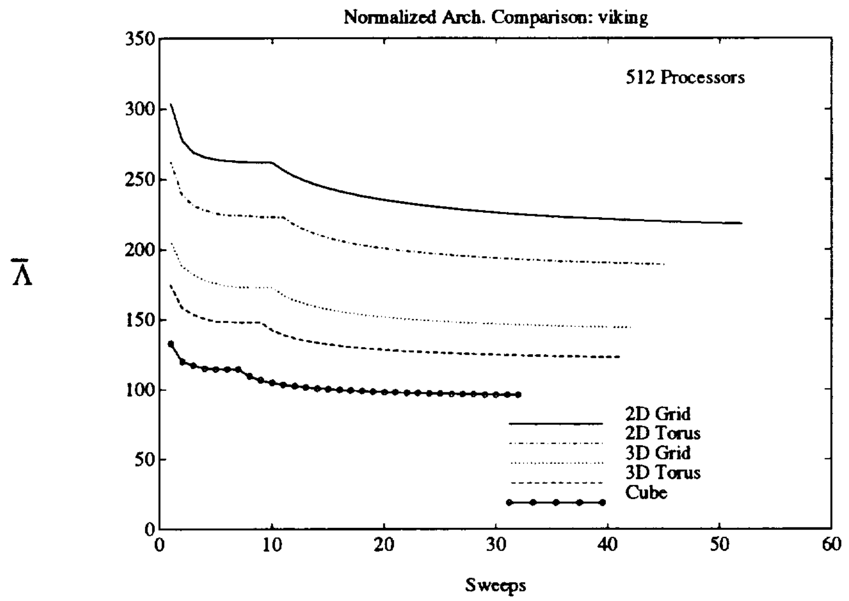


Figure 3.27: Comparison of communication capabilities for 5 interconnection schemes using the viking test case mapped to 512 processors.

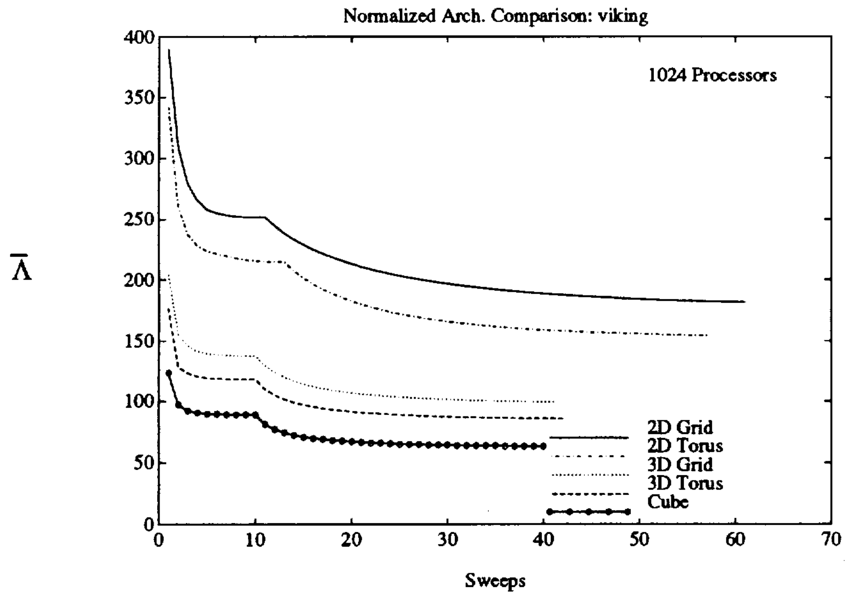


Figure 3.28: Comparison of communication capabilities for 5 interconnection schemes using the viking test case mapped to 1024 processors.

Proc.	$\bar{\Lambda}$	Cycles	Θ
256	47.83	122	0.385
512	33.27	82	0.405
1024	23.18	52	0.445

Table 3.7: The value of Θ for the bracket test case mapped to varying size hypercubes.

In Table 3.7 we show the value of $\bar{\Lambda}$, the number of message cycles, and the value of Θ for the bracket test case mapped to three different hypercube systems. We used the communication compiler to compute the number of message cycles. We see that Θ increases slightly as the number of processors increases. Also, for the 1024-processor hypercube, $\bar{\Lambda}$ is about twice as large for the 2-D torus as it is for the hypercube. Therefore, for the 2-D torus to have a comparable communication time for this problem its value of Θ should be twice the value of Θ for the hypercube, 0.89 or 89% wire utilization.

The communication compiler only supports hypercubes and only handles task graphs with $\varepsilon_T < 10^6$, so, at this time, we cannot determine the actual number of message cycles for the viking test case or for the other systems. However, we can make relative comparisons. In Figure 3.25 we see that on the 2-D torus and 3-D torus, $\bar{\Lambda}$ is 56.08 and 29.25 for the bracket test case. The ratio of these is approximately two-to-one. Therefore, if the value of Θ for the 3-D torus is greater than 0.52, then the many-to-many operation for this case on the 2-D torus will always take longer than on the 3-D torus. The situation is similar for the viking test case on 1024 processors. The value of $\bar{\Lambda}$ is 154.3 and 85.47 for the 2-D torus and 3-D torus. Therefore, if the value of Θ for the 3-D torus is greater than 0.55, then the many-to-many operation for this case on the 2-D torus will always require more time than the same operation on the 3-D torus.

Dally [21] analyzes *k-ary n-cubes* with fixed wire budgets for three applications: shortest path, Max-Flow, and graph partitioning. He concludes that low-dimensional networks need less time to support the communication requirements of his applications than high-dimensional networks.

At present we do not have sufficient data to determine whether or not Dally's conclusions for has applications apply to the applications studied here. However, we conjecture that that the communication requirements of these 3-D unstructured grid problems are better satisfied by a parallel computer whose processors are interconnected in a 3-D grid or 3-D torus compared to processors with a 2-D grid or 2-D torus interconnection. This is an area of future work.

3.10 Summary

We have introduced CPE, a parallel pairwise exchange heuristic for approximately solving the mapping problem. We showed that reducing the number of message cycles reduces the communication time. We also demonstrated that there is good correlation between $\bar{\Lambda}$ and message cycles validating our choice of Λ_1 as the objective function. Additionally, we showed that for our test cases, and RSB initial mapping followed by CPE achieves better mappings than random, naive, or gray code initial mappings followed by CPE. RSB followed by CPE is compared with SA for three test cases and shown to result in better mapping for two test cases but at a substantial time savings.

Finally, we showed that one can use CPE to make a relative comparison of the communication capabilities of various interconnection schemes for a set of hypothetical, but plausible, parallel computers. However, a comprehensive study is beyond the scope of this thesis.

CHAPTER 4

Massively Parallel Euler Solver for 2-D Unstructured Grids

In this chapter we develop a data parallel implementation of a computational fluid dynamics code and discuss how CPE is used to map tasks to processors to reduce the communication time. This flow code is the mesh-vertex upwind finite-volume scheme for solving the Euler equations on 2-D triangular unstructured meshes developed by Barth and Jespersen [5]. It has been implemented on the Cray-2 [5] and Cray-YMP. The implementation described here performs computations identical to the code developed for vector processing but has been restructured for efficient massively parallel execution on the CM-2.

First, a description of the algorithm is given. We then direct the edges of the discretizing mesh. The directed edge information is used to group data within tasks. We show that this reduces the amount of communication by a factor of two and leads to an optimal load balance. Also, we discuss the use of CPE in the flow code and compare the performance of this application with and without CPE. We show that using CPE to map tasks to processors reduces the communication time by a factor of 2.23. The result is a load-balanced compute-bound parallel implementation of the Euler code. Finally, the performance of this flow code running on 8K processors of the Connection Machine CM-2 is compared with a similar code executing on one processor of a Cray-YMP and a 64-node and 128-node Intel iPSC/860.

4.1 Mathematical Background and Algorithm Description

Here we give a brief discussion of the algorithm used in the flow solver. A detailed description is provided by Barth and Jespersen [5] and Hammond and Barth [34]. Consider the integral form of the Euler equations of gas dynamics in a

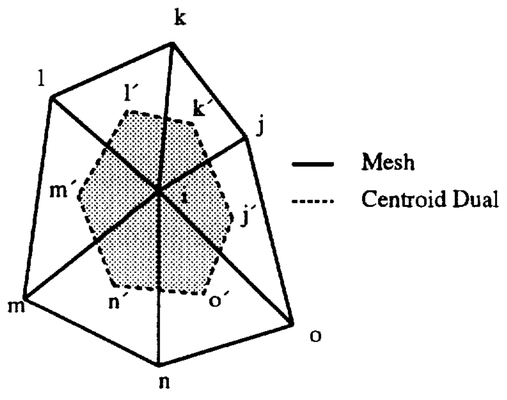


Figure 4.1: A small triangular mesh and one control volume.

general region Ω with boundary $\partial\Omega$:

$$\frac{d}{dt}\int_{\Omega}\mathbf{u}\,da+\int_{\partial\Omega}\mathbf{f}(\mathbf{u},\mathbf{n})\;dl=0. \tag{4.1}$$

In this equation $\mathbf{u}$ is the 4-vector of conserved variables: mass, momentum in each dimension, and energy. The vector function $\mathbf{f}(\mathbf{u},\mathbf{n})$ is the flux of $\mathbf{u}$ through a surface with normal orientation $\mathbf{n}$. In developing a finite-volume scheme, the integral form of the Euler equations is used for a tessellation $\mathcal{T}(\Omega)$ of Ω comprised of disjoint control volumes (also called *cells*) c_i such that $\cup c_i = \Omega$. Here, Ω is discretized with triangles and $\mathcal{T}(\Omega)$ is formed by connecting the centroids of neighboring triangles. For example, Figure 4.1 shows a portion of a triangular discretization. There are six triangles, with solid line edges, meeting at i. The centroids of the triangles are joined with dashed lines. The shaded region is the control volume c_i corresponding to mesh vertex i. Figure 4.2 shows the control volume tessellation of the region

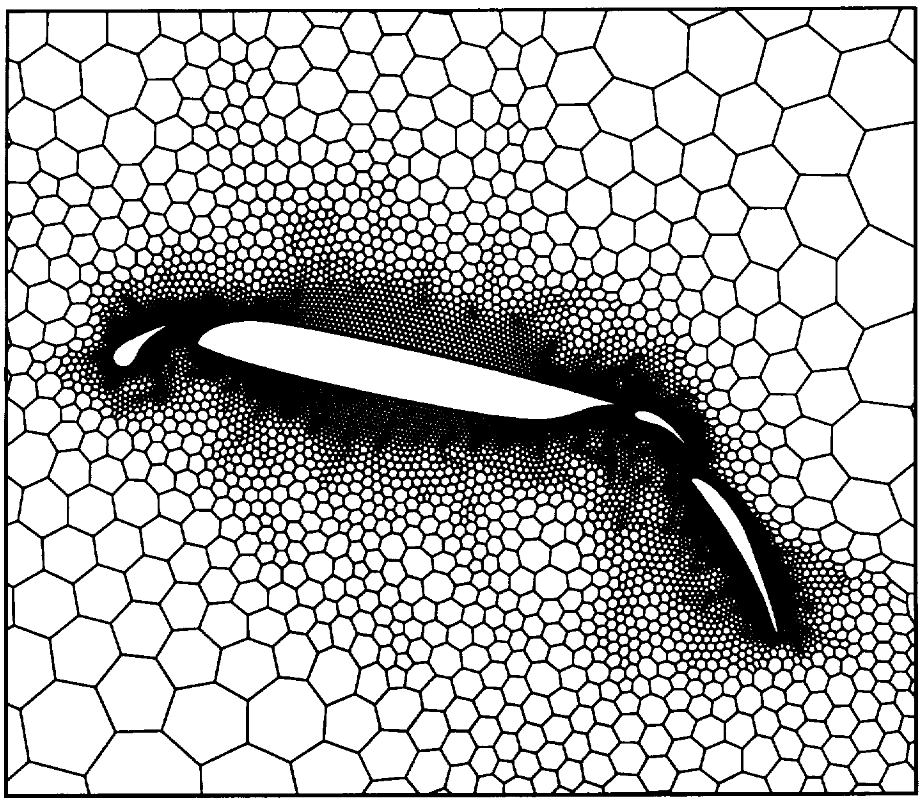


Figure 4.2: Centroid dual constructed by connecting adjacent triangle centroids.

surrounding the four element airfoil in test case 4elt, shown in Figure A.4.

Applying (4.1) to each control volume yields

$$\frac{d}{dt}\int_{c_i} \mathbf{u}\; da + \int_{\partial c_i} \mathbf{f}(\mathbf{u},\mathbf{n})\; dl = 0. \tag{4.2}$$

Fundamental to the method is the definition of the integral cell average, $\overline{\mathbf{u}}$

$$(\overline{\mathbf{u}}A)_{c_i} = \int_{c_i} \mathbf{u}\; da, \quad A_{c_i} = \int_{c_i} da. \tag{4.3}$$

Using (4.3), (4.2) is rewritten

$$\frac{d}{dt}(\overline{\mathbf{u}}A)_{c_i} + \int_{\partial c_i} \mathbf{f}(\mathbf{u},\mathbf{n})\; dl = 0. \tag{4.4}$$

In the higher order extension of Godunov's scheme [33] and the extension considered here, the integral cell averages of **u** are the fundamental unknowns. It applies to arbitrary shaped cells using piecewise linear distributions of **u** in each cell. The linear distribution of any component of **u** in cell c_i expanded about its centroid (x_{c_i}, y_{c_i}) is denoted by

$$u(x,y)_{c_i} = \overline{u}_{c_i} + (\nabla u)_{c_i} \cdot [x - x_{c_i}, y - y_{c_i}]. \tag{4.5}$$

By the use of centroidal coordinates, we restate (4.3) component-wise as

$$\int_{c_i} u(x,y)_{c_i}\, da = \overline{u}_{c_i} \int_{c_i} da = (\overline{u}A)_{c_i}. \tag{4.6}$$

The solution unknowns are approximate *pointwise* values of the solution located at the centroid of each control volume and they are associated with vertices of the mesh.

Two distinct values of the solution can be obtained along a cell boundary, because the piecewise polynomials are discontinuous from cell to cell. To resolve this, the Euler flux is supplanted by a "numerical flux function", $\overline{\mathbf{f}}(\mathbf{u}^+, \mathbf{u}^-, \mathbf{n})$, which when given these two solution states $\mathbf{u}^+$ and $\mathbf{u}^-$ produces a single unique flux. The numerical flux function is derived from approximate solutions to the Riemann problem of gas dynamics. In the computations, an approximate solver developed by Roe [57] is used. Approximating (4.4) by piecewise polynomials and a numerical flux function yields

$$\frac{d}{dt}(\overline{\mathbf{u}}A)_{c_i} + \int_{\partial c_i} \overline{\mathbf{f}}(\mathbf{u}^+, \mathbf{u}^-, \mathbf{n})\, dl = 0. \tag{4.7}$$

To complete the discretization of the flux integral, we note that ∂c_i is composed of straight line segments and perform a midpoint quadrature evaluation where (ξ_e, η_e) denotes the midpoint of an edge e of control volume c_i,

$$\frac{d}{dt}(\overline{\mathbf{u}}A)_{c_i} + \sum_{e \in \partial c_i} \overline{\mathbf{f}}(\mathbf{u}^+(\xi_e, \eta_e), \mathbf{u}^-(\xi_e, \eta_e), \mathbf{n}) l_e = 0. \tag{4.8}$$

An important task in this solution process is the calculation of the piecewise linear solution distribution in each control volume given solution unknowns at vertices of the mesh. In the case of linear distributions, the linear functions must be exact whenever the true solution varies linearly over the support of the cell discretization (distance-one neighbors of the mesh). To accomplish this task, a numerical approximations to the exact Green-Gauss formula is used,

$$\int_{\Gamma} \nabla u \, da = \int_{\partial\Gamma} u\mathbf{n} \, dl, \tag{4.9}$$

for some path $\partial\Gamma$ surrounding c_i. For linear functions the gradient is constant in Γ

$$\int_{\Gamma} \nabla u \, da = (\nabla u)_{\Gamma} A_{\Gamma}.$$

Using pointwise values of the unknowns at vertices of the mesh, choose a path connecting distance-one neighbors of the mesh (see Figure 4.1). A trapezoidal quadrature formula for the integration of the right-hand-side of (4.9), guarantees that linear reconstruction is exact whenever the function varies linearly over the support of the reconstruction. Using some algebraic manipulations (see Barth [4]), the trapezoidal integration about the path $\partial\Gamma_i$ can be rewritten as

$$(\nabla u)_{c_i} = \frac{3}{A_{\Gamma}} \sum_{\alpha} \frac{1}{2}(u_{\alpha} + u_0)\, \mathbf{n}_{\alpha}\, l_{\alpha}, \qquad \forall \alpha \in \{j, k, l, m, n, o\}, \tag{4.10}$$

where $\mathbf{n}_{\alpha}$ is the normal of the control volume associated with the edge of the centroidal dual. For example, in Figure 4.1, when $\alpha = j$ then $\mathbf{n}_{\alpha}$ and l_{α} are the normal and length of edge (j', k').

To summarize the previous discussion, the solution process consists of three primary steps:

(i) **Gradient Calculation in Each Control Volume**: Given solution unknowns, construct monotone piecewise linear polynomials for use in (4.10).

(ii) **Flux Evaluation on Each Edge**: For each edge in $\mathcal{T}(\Omega)$, perform a flux quadrature consistent with linear functions, (4.8).

(iii) **Evolution in Each Control Volume**: Collect flux contributions in each control volume and evolve in time using a 4-stage Runge-Kutta numerical integration scheme.

4.2 Edge Direction and Data Storage

In this section, we show that to achieve efficient parallel computation, it is not sufficient to simply identify a commensurate number of tasks to be executed in parallel by the processors, one must also determine how to group the data within tasks to reduce the communication overhead. In particular, we direct the edges of the unstructured mesh to determine how to group data associated with the vertices (vertex data) and the data associated with the edges (edge data) together within each task. In the massively parallel implementation of the flow solver described above, we associate a task with each grid point (recall that each grid point is also associated with the centroid of a control volume). Let task u be associated with grid point i and task v be associated with grid point j. If i is adjacent to j in the discretizing grid, then $\langle u, v\rangle, \langle v, u\rangle \in E_T$.

We now have to decide which data is stored by each task. In a mesh-vertex scheme, solution variables (mass, momentum, and energy) and the areas of the control volumes are associated with each vertex in the mesh. The lengths and unit normals of the edges of the control volumes are associated with each edge of the mesh.

Assume that $\nu_T \geq \nu_P$. Clearly, every task contains the data for one vertex and the data for several edges. In Figure 4.3 we illustrate the way that the data is stored. The figure shows an array from 1 to ν_T of large rectangles representing the data associated with the vertices and a short array below each array element representing the data from a few edges. A large rectangle and the array of smaller rectangles below it represents the data stored in one task.

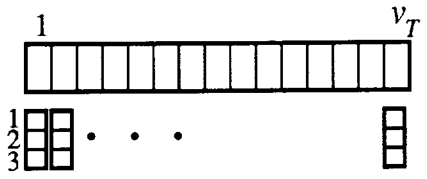


Figure 4.3: Storage of edge data and vertex data.

The three steps of the algorithm above are either performed once per edge (i and ii) or once per vertex (iii). Although the discussion below focuses on the flux calculation, one should notice the similarity between (4.8) and (4.10). In each one, some function of the edge and vertex data is evaluated once for each edge of the control volume and the values for the edges surrounding each vertex are summed to be stored as vertex data. Therefore, the same arguments that are made about the distribution of data for the flux calculations apply to the distribution of data used in the gradient calculations as well.

Each task performs flux computation at edges of the control volumes which surround the associated grid point. Implementations on sequential and vector computers typically perform this computation in a loop executed once for each edge of the control volume. The ε_T flux calculations can be done in parallel, so each task holding edge data can perform the corresponding flux calculation. Unnecessary communication is required if the edge data and vertex data for a flux calculation are each stored in different tasks. Recall that the CM-2 is a SIMD computer and that all of the processors perform the same operation at the same time. Thus, as the flux calculations are performed, if one task holding some edge data needs to gather information from two tasks holding the necessary vertex data, and the two tasks are mapped to different processors, then all tasks need to communicate. Also,

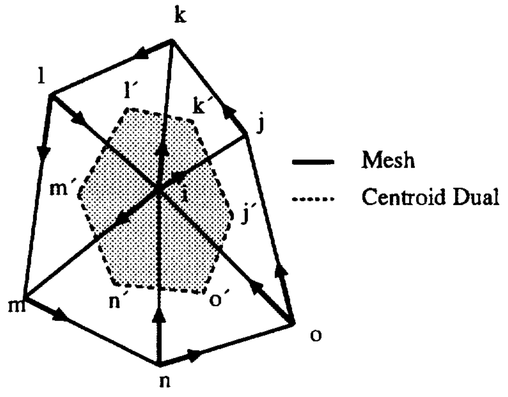


Figure 4.4: Control volume and directed edges on a small triangular mesh.

the resulting flux calculation contributes to the vertex data in two control volumes so the task must then distribute its results back to the two tasks from which it received data. This requires a total of $4\varepsilon_T$ communications. For example, consider computing the flux through $\langle k', j'\rangle$ in Figure 4.1. Suppose that the vertex data associated with vertices i and j are stored by tasks u and v respectively and that the edge data associated with $\langle k', j'\rangle$ is assigned to some task x. Any of the three tasks could compute the flux through this edge of the control volume. Let task x do the computations. Tasks u and v send their conserved values to x, the flux is computed, and the results are then sent back to tasks u and v to be accumulated.

Now, suppose that we direct the edges of the mesh from Figure 4.1 as shown in Figure 4.4. When some edge of the mesh $\langle i, j\rangle$ is directed from i to j, then the task associated with grid point i stores the vertex data for i and the edge data corresponding to the control volume edge associated with $\langle i, j\rangle$, in this case $\langle j', k'\rangle$. In Figure 4.4, the mesh has directed edges from vertex i to vertices m, k, and j.

Therefore, the task storing the vertex data for grid point i also stores the edge data for $\langle j', k' \rangle$, $\langle k', l' \rangle$, and $\langle m', n' \rangle$. During (ii) each task performs one flux computation for each control volume edge associated with the outward directed edges of the grid point. For each of the outward directed edges, every task does the following: it receives vertex data from one other task, performs a flux computation, and then sends results back to the same task. Every task has two thirds of the information needed for the flux computation stored *locally.* For example, in Figure 4.4 there is a directed edge from i to j. Therefore, the flux across $\langle k', j' \rangle$ is computed by the task storing the vertex data from grid point i. For the flux calculation, each task executes a one-to-many communication to distribute its conserved values to tasks storing neighboring vertices joined by inward directed edges of the mesh. It performs the appropriate flux computations and then performs a many-to-many communication primitive to the tasks storing vertices joined by outward directed edges of the mesh. For example, the task storing data from grid point i will receive data from the tasks storing vertices j, k, and m and send data (one-to-many) to the tasks storing vertex data for n, o, and l. After the task storing data for vertex i computes the flux through $\langle j', k' \rangle$, $\langle k', l' \rangle$, and $\langle m', n' \rangle$ it sends the results (many-to-many) to the tasks associated with grid points j, k, and m and receives results from the tasks associated with grid points n, o, and l. Therefore, with the data stored in this manner, each task performs 2 communications for each of its outward directed edges, for a total of 2ε. This is half the number of communications required otherwise.

Using the edge direction to determine which task computes the flux through each edge reduces the communication by half. But, edge direction can result in a load imbalance if one vertex has many more outward pointing edges than the other vertices. Recently, Chrobak and Eppstein [17] have given a linear time algorithm for orienting the edges of any planar graph G such that $\delta^{+}(G) = 3$. Therefore, no task needs to compute the flux across more than three edges of the control volume. Directing the edges in this manner is an optimal load balance.[1]

Now, we prove that using the directed edge information to group vertex and edge data within tasks leads to an efficient parallel implementation. We show that the serial code is $O(\varepsilon)$ and that with p tasks, our parallel algorithm is $O(\frac{\varepsilon}{p})$. It is clear that for step (iii) above, that the storage scheme requires no redundant calculations since these computations are done once for each vertex of the mesh and we have assigned one task to each vertex. However, steps (i) and (ii) must be performed once for each edge of the control volume. It is obvious that a serial implementation will perform one flux computation and one gradient computation for each edge of the mesh. Recall that two vertices of the mesh share a control volume edge and that we direct the edges of the mesh to determine which task performs the flux computation. We claim that there are approximately three times as many edges as vertices in a triangular discretization of 2-space. Therefore, if each task associated with a vertex performs three flux calculations then the proposed data storage scheme completes the $O(\varepsilon)$ work in $O(\frac{\varepsilon}{p})$ time with p tasks.

We prove the following:

Proposition: For a 2-D mesh of triangular elements, $\varepsilon = 3\nu - 6 - \sum_{f \in \mathcal{F}_b(G)}(\delta(f) - 3)$.

Proof: Let G be a planar graph. Recall that Euler's formula for planar graphs (1.2) states that

$$\nu - \varepsilon + \phi = 2.$$

For any planar graph, each edge is shared by 2 faces. Therefore, the sum of the degree of all of its faces is equal to twice the number of edges:

$$\sum_{f \in \mathcal{F}(G)} \delta(f) = 2\varepsilon.$$

[1]For nonplanar graphs, bounding the outdegree by a constant is not possible. It is possible to bound the outdegree to be equal to one half of the maximum degree of the graph. In general, tighter bounds are not known for nonplanar graphs.

We can subtract 3 from each term in the summation and 3ϕ from the right to get

$$\sum_{f\in\mathcal{F}(G)} (\delta(f)-3) = 2\varepsilon - 3\phi$$

or

$$\sum_{f\in\mathcal{F}_b(G)} (\delta(f)-3) = 2\varepsilon - 3\phi, \tag{4.11}$$

since $\forall f \in \mathcal{F}(G) \setminus \mathcal{F}_b(G)$, $\delta(f) = 3$. Rearranging (1.2) and substituting into (4.11) yields

$$\sum_{f\in\mathcal{F}_b(G)} (\delta(f)-3) = -\varepsilon + 3\nu - 6. \tag{4.12}$$

Solving for ε yields

$$\varepsilon = 3\nu - 6 - \sum_{f\in\mathcal{F}_b(G)} (\delta(f)-3).$$

■

Therefore, if the number of boundary edges is small relative to the number of interior edges then, asymptotically, there are three times as many edges as vertices in a triangular mesh. In the 4elt test case, $\nu = 15606$, $\varepsilon = 45878$, $\phi = 30269$, and $\sum_{f\in\mathcal{F}_b(G)} \delta(f) = 949$.

4.3 Fast Communication

In the previous section, we showed that using the directed edge information to group edge and vertex data to be stored together in a task results in 2ε communications. In this section, we show that we can reduce the amount of time required for these 2ε communications by using CPE to map tasks to processors. The communication compiler is used to schedule the communications on the wires.

During the flux and gradient calculations, the initial gathering of information is accomplished using a one-to-many operation since the same data is sent to all neighboring tasks corresponding to the inward directed edges of the mesh. After

Operation	Naive	Naive & CPE	RSB & CPE	No Comm.
one-to-many	4.055	2.210	1.738	-
gradient comp.	22.454	13.745	12.685	5.574
flux comp.	23.951	19.490	18.864	16.049
many-to-many	9.706	5.064	4.242	-
boundary comp.	5.494	4.583	4.739	3.178
total	65.885	45.092	43.223	24.801

Table 4.1: Time in seconds for 400 iterations of kernel in unstructured Euler code.

the calculation, the distribution of results uses a many-to-many operation since unique data is sent to each task corresponding to an outward directed edge.

In Table 4.1 we compare the performance of this code on the CM-2 for test case 4elt using a naive mapping, a naive initial mapping followed by CPE, the RSB initial mapping followed by CPE, and just the computation. The code is written in *lisp and timings were done using a Sun4/490 front end running version 6.0 of the CM operating system. Time is given in seconds for 400 iterations of each operation, corresponding to 100 iterations of the flow solver using a 4-stage Runge-Kutta numerical integration scheme. In the trial using RSB, the quotient graph is embedded using a naive embedding followed by CPE to improve the embedding. Next, the grouping of vertices into partitions was relaxed, and CPE was used to improve the mapping. During the mapping process, no special considerations were made to process the subset of vertices that communicate during the computations and communications for the boundary conditions. In the last case, the communication calls are commented out in the code. The sum of the times taken by the individual components is larger than if one simply runs all of the pieces together. We see that RSB followed by CPE reduced the communication time 2.23 times compared to a naive mapping. Also, the last column shows that the time spent computing is 24.801 seconds and the total time with the RSB initial mapping and CPE is 43.223 seconds.

This means that 18.422 seconds, or 42.6% of the time was spent communicating and 57.4% of the time was computing.

4.4 Timing and Results

We compare the performance of the unstructured flow solver on 8K processors of a Connection Machine CM-2 with one processor of a Cray-YMP. Note that this is $\frac{1}{8}$ of each of the full machines. The code on the CM-2 is an implementation of the vertex-based scheme of the mesh-vertex scheme with piecewise linear reconstruction and 4-stage Runge-Kutta integration used for evolution in time. As in the timings above, the code is written in *lisp and timings were done using a Sun4/490 front end running version 6.0 of the CM operating system. The calculations in the comparison are all done in 32-bit arithmetic since the CM-2 we used did not have 64-bit hardware at the time the code was developed. The geometric calculations for edge lengths, edge normals, control volume areas, etc., were all precomputed in 64-bit arithmetic on the CM-2 and stored as 32-bit values. This was necessary to obtain accurate gradients used in linear reconstruction and probably indicates the need for 64-bit precision for this type of computation. This will be especially important for viscous flow calculations where the control volume areas will be orders of magnitude smaller. The 64-bit calculations on the CM-2 were computed in software and the initialization was not timed as part of the benchmark on either machine.

The calculations performed on the Cray-YMP are identical to the ones performed on the CM-2. It is written in Fortran and all computations are in 64-bit arithmetic. Also, the clock period on the YMP is 6 ns. We used the flow tracing package *perftrace* to analyze the Cray code to determine the floating point usage. There are approximately 300 floating point operations per edge, per iteration of the flow solver. The edges of the mesh were colored and the code is vectorized over edges of the dual grid utilizing gather-scatter. The shortest vector has length 500.

Computer	Processors	Mflops	Time in Seconds
CM-2	8192	136	43
Cray Y-MP	1	150	39
Intel iPSC/860	64	188	31

Table 4.2: Mflops and Time in seconds for unstructured Euler code on three systems.

The main part of the code consumes 97% of the time and sustains 150 Mflops.

As mentioned earlier, the test case used has 15606 vertices, 45878 edges, 30269 faces, 4 bodies, and 949 boundary edges. Note that since there are approximately twice as many vertices as processors on the CM-2, two tasks are assigned to each processor[2]. The full grid for this test case is shown in Figure A.3 and a close up of the airfoil is shown in A.4.

The flow was computed at a Mach number of 0.1 at 0 degrees angle of attack relative to the mesh. On one processor of a dedicated YMP, 100 time steps of the code took 39 seconds. On 8K processors of the CM-2, the same computation takes 43 seconds.

The same algorithm has been developed to run on the Intel iPSC/860 hypercube by Venkatakrishnan, Simon, and Barth [80]. RSB was used to partition the task graph and a naive embedding was used for the timing. They compared different schemes for embedding the quotient graph into the hypercube. They found that the difference in the amount of time for communication was negligible for several different embedding schemes.

In Table 4.2, we show the performance of this code on the Cray, the Intel, and the CM-2. We conclude that using CPE to map the tasks to processors has made the performance of the unstructured Euler code on a data-parallel SIMD computer

[2]The code runs at a "VP-ratio" of 2

similar to the performance achieved on one processor of a Cray Y-MP and a 64-processor Intel iPSC/860.

CHAPTER 5
Sparse Matrix-Vector Products

In this chapter we study techniques for multiplying sparse matrices and vectors. This is a kernel operation in the solution of large sparse linear systems of equations by iterative techniques such as the Krylov subspace methods. Also, it is the second application in which we show how CPE is used to reduce the communication time.

Given $x \in \Re^n$ and sparse $A \in \Re^{n \times n}$, compute $y \in \Re^n$

$$y = Ax. \tag{5.1}$$

We compare three methods for computing (5.1). Each method is described and a simple example is given. Finally, we give times for each of the methods for several test cases. The first method, scan-based, uses scan operations (scans) and each virtual processor stores one nonzero element of the matrix. The second method, column-wise, is based on storing the nonzero elements of a column of the matrix in each virtual processor. The third method, row-wise, stores the nonzero elements of a row of the matrix in each virtual processor. For the last two methods, we associate a task with each column and row of the matrix, respectively. If a_{ij} is nonzero, then $\langle i, j \rangle \in T$. We use CPE to map tasks to processors and show that the communication time is reduced by approximately a factor of two for our test cases. We demonstrate that, on the CM-2, row-wise sparse matrix-vector multiplication mapped with CPE is an order-of-magnitude faster than scan-based operations for our test cases.

Saltz *et al.* [62] compare a scan-based and a row-wise sparse matrix vector multiplication scheme for matrices arising from several synthetic grids and several banded matrices. The synthetic grid test cases are generated from a 5-point finite difference stencil on a regular grid except that with probability 0.2, each edge of the stencil was changed from joining a grid point and its neighbor to joining a grid point and a randomly chosen point in the grid. The grid sizes they use are: (64×64), (64×128), (128×128), (256×128), and (256×256). The second test cases are square, banded matrices, with bandwidths of 4, 8, and 16, and between 4K and 64K rows. In the row-wise operation, one row of nonzero elements is assigned to each virtual processor, however, they do not say how the rows are assigned to the processors. The communication compiler is used for the communication in the row-wise case. On a 16K CM-2, the row-wise sparse matrix vector multiplication scheme is faster than the scan-based scheme in all but the smallest cases. When the number of rows or columns in the matrix is less than the number of processors, the scan-based operation is faster for their test cases. Here we use data from applications rather than banded or synthesized matrices for our comparisons since the banded and synthesized matrices are representative of the class of problems we study here. Also, we use CPE to map the associated task graphs to the processors to reduce the communication time.

5.1 Scan-based Sparse Matrix-Vector Multiplication

Iverson first introduced *scan operations* as part of APL [39]. Scans take as input a binary operator $\oplus$ with identity i and an ordered set $b = [b_0, b_1, \ldots, b_{n-1}]$. We assume that each b_i is assigned to a different processor. Exclusive scans return the ordered set $[i, b_0, (b_0 \oplus b_1), (b_0 \oplus b_1 \oplus b_2), \ldots, (b_0 \oplus b_1 \oplus \cdots \oplus b_{n-2})]$. Inclusive scans return $[b_0, (b_0 \oplus b_1), (b_0 \oplus b_1 \oplus b_2), \ldots, (b_0 \oplus b_1 \oplus \cdots \oplus b_{n-1})]$. Another form of scan operation is the backward scan (right to left) which returns $[(b_0 \oplus b_1 \oplus \cdots \oplus b_{n-1}), (b_1 \oplus \cdots \oplus b_{n-1}), \ldots, (b_{n-2} \oplus b_{n-1}), b_{n-1}]$.

Another variant of scans are *segmented scans* [52, 67] which enable one to execute scans independently on contiguous subsets of the input, called segments. Segmented scans take an additional argument, a set of segment flags that have the

b	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
segment	T	F	F	F	F	F	F	T	F	F	F	F	F	F	F
								add-scan							
unseg.	1	3	6	10	15	21	28	36	45	55	66	78	91	105	120
seg.	1	3	6	10	15	21	28	8	17	27	38	50	63	77	92
								copy-scan							
unseg.	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
seg.	1	1	1	1	1	1	1	8	8	8	8	8	8	8	8

Figure 5.1: Example of inclusive add-scan and copy-scan operations, unsegmented and segmented versions.

same number of elements as the input set b. The input flags have value T or F. If the i^{th} segment flag has value T, then b_i is the beginning of a segment.

Figure 5.1 shows an example of inclusive add-scan and copy-scan operations in both segmented and unsegmented mode applied to b. The source set b and the segment are shown at the top. The segment flags indicate that b_0 and b_7 start new segments.

Blelloch [11] shows how scan primitives are used in algorithm design and how they can be implemented in hardware. He shows that a scan operations is no more expensive than a reference to shared memory. Also, the complexity of many algorithms developed for the EREW PRAM model of computation can be reduced by $O(\log n)$ when implemented with scans.

To compute (5.1) using scans, the nonzero elements of A are stored in a linear array **a** in row major order. Suppose that there are $nz(A)$ nonzero elements in A. Five additional $nz(A)$-element arrays are needed. The first, `row`, is an array of the row numbers of the elements of A. The second array, `col`, contains the column numbers of the corresponding elements of **a**. The third, `seg`, is a segment array whose elements are T if the corresponding element in **a** is the first nonzero element in a row of A. Thus, the nonzero elements of the rows of A are stored in segments.

The fourth and fifth arrays are temporary arrays, `x_temp` and `y_temp`.

A scan-based algorithm is currently used in the sparse matrix vector multiplication on the Connection Machine CM-2 in version 2.2 of the Scientific Subroutine Library [76]. It works as follows:

1. distribute elements of x,
2. parallel element-wise multiplication,
3. backwards segmented add_scan,
4. send results back to y.

Here we give an example of multiplying a sparse 5×5 matrix A by a vector x. Fortran90 statements corresponding to each operation are also shown in []'s. Let

$$A = \begin{pmatrix} 1 & 0 & 5 & 0 & 6 \\ 0 & 1 & 2 & 3 & 0 \\ 2 & 2 & 3 & 0 & 2 \\ 3 & 0 & 3 & 0 & 3 \\ 1 & 0 & 0 & 0 & 5 \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}. \tag{5.2}$$

The arrays `x`, `a`, `row`, `col`, and `seg` are below.

`x`=	{1	2	3	4	5}										
`a`=	{1	5	6	1	2	3	2	2	3	2	3	3	3	1	5}
`row`=	{1	1	1	2	2	2	3	3	3	3	4	4	4	5	5}
`col`=	{1	3	5	2	3	4	1	2	3	5	1	3	5	1	5}
`seg`=	{T	F	F	T	F	F	T	F	F	F	T	F	F	T	F}

Each array is distributed one element per processor.

The first step in computing a scan-based sparse matrix-vector multiplication is to distribute the elements of `x` to the elements of `temp_x`. Each virtual processor storing a nonzero element from column i gets a copy of `x[i]`. Since `x` is the result of prior computations, it is typically not stored in this distributed manner. For example, in the case of solving large sparse linear systems of equations by the conjugate gradient method [36], **x** is a search direction and **y** is used to scale an update of the approximate solution at each iteration.

Executing [`x_temp(:) = x(col(:))`] results in

`x_temp=` {1 3 5 2 3 4 1 2 3 5 1 3 5 1 5}.

Next, `a` and `x_temp` are multiplied element-wise and the result is stored back in `x_temp` [`x_temp = x_temp * a`].

We now have

`x_temp=` {1 15 30 2 6 12 2 4 9 10 3 9 15 1 25}.

Next, we perform an inclusive add-scan operation with the following inputs: `y_temp` is the destination, `x_temp` is the source data, `seg` specifies the beginning of the rows, and `backward` specifies the direction.

[`call add_scan(y_temp, x_temp, seg, backward)`],

`y_temp=` {46 45 30 20 18 12 25 23 19 10 27 24 15 26 25}.

Finally, the elements of `y_temp` at the end of the backward segments are are sent to `y` [`where (seg) y(row(:)) = y_temp(:)`].

`y =` {46 20 25 27 26}

Randomized routing has also been considered. For an n-cube, suppose that every node contains one message to send to a distinct node. Valiant and Brebner [79] and Valiant [78] describe a distributed randomized algorithm that can route every message to its destination without two messages passing down the same wire at the same time in $O(n)$ with very high probability for all such message routing requests. The algorithm consists of two phases run consecutively. The first phase sends each message to a randomly chosen node. For every message, every node has the same probability of being chosen, and the choices are independent of each other. The second phase then routes each message from the random intermediate location to its final destination.

This approach is an option on the Connection Machine in the CMSSL library function and is chosen by setting a the `irandom` parameter to 1. At the end of this chapter, we show that randomization reduces the communication time compared to non-randomized sparse matrix-vector multiplication, however, it is still much slower the column-wise and row-wise methods.

5.2 Column-wise Sparse Matrix-Vector Multiplication

One can also compute (5.1) by forming a linear combination of the columns. Each processor stores some x_j, y_j, the nonzero elements of column j of A and an array of row numbers corresponding to the elements of A. The column-wise from of sparse matrix vector multiply is thus:

1. local multiply: in parallel, each processor multiplies its x_j with each nonzero element of the column.

2. send products: the result of $(A_{ij} * x_j)$ is sent to the processor holding y_i.

3. sum partial results: each processor holding some y_i adds $(A_{ij} * x_j)$, $1 \leq i \leq n$, from all the processor holding nonzero elements of row i.

For the example, we use the same A and x from (5.2). Suppose that we have five processors and that the data associated with column j gets assigned to processor j. The data stored in each processor is shown across a row in the table. Initially, we have

proc.	a	row	x
1	{1 2 3 1}	{1 2 3 5}	1
2	{1 2}	{2 3}	2
3	{5 2 3 3}	{1 2 3 4}	3
4	{3}	{2}	4
5	{6 2 3 5}	{1 3 4 5}	5

Multiplying the nonzero elements in each processor by the element of x yields:

proc.	y_temp
1	{1 2 3 1}
2	{2 4}
3	{15 6 9 9}
4	{12}
5	{30 10 15 25}

Finally, we send the partial results and sum to form y.

proc.	y_temp	y
1	{1 15 30}	46
2	{2 6 12}	20
3	{9 2 4 10}	25
4	{3 9 15}	27
5	{25 1}	26

5.3 Row-wise Sparse Matrix-Vector Multiplication

Finally, we consider computing (5.1) using inner products. Each processor stores the following data: some x_i, y_i, the nonzero elements of the i^{th} row of A, and the column numbers corresponding to the nonzero elements of the i^{th} row of A. The elements of x are then sent to the appropriate processors and each processor performs a local inner product to calculate y. The row-wise from of sparse matrix vector multiply is:

1. get x: each processor gets the x_i corresponding to the column numbers of the row numbers of the nonzero elements it stores.

2. inner product: each processor performs an inner product of the x's it has received with the nonzero elements of the row it holds.

For the example, we use the same A and x from (5.2). Suppose that we have five processors and that the data associated with column i gets assigned to processor i. Initially, we have

proc.	a	col	x
1	{1 5 6}	{1 3 5}	1
2	{1 2 3}	{2 3 4}	2
3	{2 2 3 2}	{1 2 3 5}	3
4	{3 3 3}	{1 3 5}	4
5	{1 5}	{1 5}	5

After each processor gets the values of x, they are stored in an array **x_temp** and perform the inner product to get y:

proc.	a	x_temp	y
1	{1 5 6}	{1 3 5}	46
2	{1 2 3}	{2 3 4}	20
3	{2 2 3 2}	{1 2 3 5}	25
4	{3 3 3}	{1 3 5}	27
5	{1 5}	{1 5}	26

5.4 Comparison of Three Methods

In this section we compare the performance of the three methods of multiplying a sparse matrix and a vector on several test cases. The matrices used for the test cases are the adjacency matrices of the task graphs for test case 3elt, 4elt, and motor.

		column-wise		row-wise	
	scans	Naive	CPE	Naive	CPE
router get	36.27	-	-	-	-
mult. time	0.27	-	-	-	-
add_scan	5.66	-	-	-	-
router send	5.03	-	-	-	-
comp.	-	1.22	1.22	0.78	0.78
comm.	-	5.62	3.02	3.02	2.26
total time	47.48	6.84	4.23	4.24	3.03
Mflops	1.16	8.02	12.99	12.96	18.10

Table 5.1: Comparison of 64-bit sparse matrix-vector multiplication schemes for 3elt task graph, time in seconds.

In Tables 5.1, 5.2, and 5.3 we show the performance of each of the sparse matrix vector multiplication techniques executed 1000 times. Timing on the CM-2 is complicated because the time of an application depends on the rate at which the CM-2 is fed instructions by the front-end computer. To try to reduce this effect, each test is run three times and the best time is used in each entry. All test cases are implemented in *lisp and run on an 8K CM-2 running version 6.0 of the operating system. The time for all entries except the last row is measured in seconds.

The total time given is not the sum of the entries in the column above it but the time taken to run the total algorithm. Compiler optimizations enable some reductions in operation thus the total time is often less than the sum of the times of the individual operations. We do not understand why the router communication time for the 4elt test case was so much larger than the time for the communication and computation time.

The heuristic mapping algorithm is used to assign the columns or rows of A and elements of x and y to the processors to minimize the communication time in the column-wise and row-wise algorithms, respectively. The communication compiler is used to communicate the elements of x in the row-wise and the partial results in

		column-wise		row-wise	
	scans	Naive	CPE	Naive	CPE
router get	121.28	-	-	-	-
mult. time	1.02	-	-	-	-
add_scan	10.32	-	-	-	-
router send	8.73	-	-	-	-
comp.	-	5.91	5.91	3.82	3.82
comm.	-	23.90	12.27	18.66	7.85
total time	140.35	29.94	18.20	22.50	11.64
Mflops	1.80	8.44	13.88	11.23	21.70

Table 5.2: Comparison of 64-bit sparse matrix-vector multiplication schemes for motor task graph, time in seconds.

		column-wise		row-wise	
	scans	Naive	CPE	Naive	CPE
router get	94.56	-	-	-	-
mult. time	1.13	-	-	-	-
add_scan	12.98	-	-	-	-
router send	13.21	-	-	-	-
comp.	-	2.46	2.46	1.56	1.56
comm.	-	12.64	6.76	7.91	4.33
total time	114.58	15.06	8.45	9.10	5.85
Mflops	1.60	12.18	21.71	20.16	31.35

Table 5.3: Comparison of 64-bit sparse matrix-vector multiplication schemes for 4elt task graph, time in seconds.

the column-wise. From *lisp, we use the command `cmi::deliver-11`. Note that the communication time in the row-wise scheme is less than the communication in the column-wise scheme. In the row-wise scheme, only one value is sent to many neighbors and the fanout optimization can be exploited.

The column labeled "router" is included for comparison with the other methods. It uses the same algorithm for multiplication as the inner product but the router is used to communicate the elements of x. Each processor makes $\Delta(T)$ calls to the *lisp instruction `pref!!`.

Scan based sparse matrix vector multiplication are appropriate when the number of rows/columns in the matrix is much less than the number of processors since it utilizes more parallelism. However, note that the time required for the add_scan alone is greater than the total time using outer and inner products in the 3elt and 4elt test cases. So, even if the time for all other operations was zero, the scan-based matrix vector multiplication would still take longer than the outer and inner product based algorithms.

There are several reasons why the row-wise algorithm is faster than the column-wise algorithm. First, the communication is faster since the elements of x are each sent to multiple destinations. This is the one-to-many communication primitive and we discussed the communication optimization that can be used to make it faster than the many-to-many communication primitive required by the column-wise storage scheme. Also, the floating point units on the CM-2 can perform a multiply and an add every cycle and the inner product operation takes advantage of this capability. In the column-wise scheme, the multiplication of matrix elements and vector elements occur at one time and the summation of products occur on different processors at a later time.

Table 5.4 compares the time for 1000 sparse matrix-vector multiplications using the CMSSL 2.2 routine `sparse_matvec_mult` called from CM Fortran and the

	CMSSL	CMSSL	Row-wise
Test Case	rand=0	rand=1	& CPE
3elt	33.70	27.56	3.03
motor	134.43	94.50	11.64
4elt	94.72	76.06	5.85

Table 5.4: Comparison of time in seconds for 64-bit CMSSL sparse matrix routines called from CM Fortran and the row-wise scheme using CPE on three test cases.

times for a row-wise sparse matrix-vector multiplication mapped with CPE in `*lisp`. Specifying rand=0 means that the nonzero elements of row one are stored in the first segment, the nonzero elements of the second row are stored in the second segment, etc. Specifying rand=1 chooses the randomization scheme described above. All computations are done in 64-bit arithmetic on an 8K CM-2 running version 6.0 of the operating system. The table shows that on the CM-2, randomization reduces the time for the scan-based sparse matrix-vector multiplication. However, sparse matrix-vector multiplication using a row-wise method and CPE is an order-of-magnitude faster than the scan-based method for the 4elt test case and slightly less than an order-of-magnitude faster for the 3elt and motor test cases. We conjecture that the row-wise form is less than an order-of-magnitude fast than the scan-based operation for the 3elt and motor test cases because there are fewer rows in the matrices than 1-bit processors in the 8K CM-2.

CHAPTER 6
Conclusions

In this thesis we have investigated the mapping problem. We developed the CPE heuristic, a highly-parallel iterative pairwise exchange algorithm in which each processor may exchange the tasks mapped to it with a small subset of the other processors. The objective function used with CPE is Λ_1 and we demonstrated good correlation between Λ_1 and the communication time on the CM-2.

For very large, very irregular problems arising in 2-D flow around complex multi-component airfoils and 3-D flow around aircraft, CPE outperformed methods based on simulated annealing – it required far less time to do the mapping and the results obtained are better. By this we mean that, for our test cases, an application requires less execution time when using a mapping produced by CPE than when using a mapping produced by SA. Compared with random and naive mappings, it reduced the communication time twofold, even for realistic, large, highly irregular, and stretched meshes.

We developed an efficient data parallel implementation of Barth and Jespersen's mesh-vertex upwind finite volume scheme for solving the Euler equations on triangular unstructured meshes [5]. An optimal edge direction was used to group vertex and edge data within a task to reduce the amount of communication by 50% in this application. CPE was used to map tasks to processors and it produced a load-balanced compute-bound computation whose throughput was slightly less than, but competitive with, a similar code on 1 processor of a Cray Y-MP and a 64 processor Intel iPSC computer.

We compared three methods of massively parallel sparse matrix-vector multiplication: scan-based, column-wise, and row-wise. We showed that mapping with CPE reduced the communication time by a factor of two for the latter two methods.

Also, we demonstrated that the row-wise method was the fastest of the three and that it achieved approximately an order of magnitude greater throughput than the scan-based method for our test cases on the CM-2.

Finally, this thesis demonstrates that a judicious assignment of tasks to processors enables data-parallel SIMD computers to efficiently solve problems that arise in the solution of discretized PDEs, where the discretizing grid is arbitrary, but static.

6.1 Future Work

There are many extensions that can be made to the work presented here. First of all, alternatives to the way that CPE exchanges messages can be studied. In some respects, CPE is the simplest deterministic-iterative algorithm; neighboring processors iteratively exchange pairs of tasks. One could consider exchanging tasks between processors other than neighbors. Also, one could try exchanging groups of tasks rather than one pair per exchange. Finally, one could allow temporary load imbalances rather than strictly enforcing each processor to hold an equal number of tasks. Other variations not mentioned here are certainly possible.

Secondly, the current implementation of CPE assumes that the vertex weights in the task graph are equal and that the edge weights in the task graph are equal. This assumption is reasonable for a SIMD computer but may not be appropriate for a MIMD system. Additionally, CPE assumes that the computer is homogeneous. CPE can be extended to operate on heterogeneous task and processor graphs.

Also, we have assumed that there are no precedence relations between the tasks. We assume that all tasks repeatedly compute and communicate, using the values of the previous communication in the current computation. One could consider extending CPE to map task graphs with precedence.

There is additional research to be done on the initial mappings. The current implementation of RSB assumes a homogeneous task graph. Currently, the entries of x_2 are sorted and the median value is chosen to balance the number of tasks in each partition. If the task weights are different, one could start at each end of x_2 and sum the values of the corresponding task weights until the sum of the task weights in each partition are roughly equal and all tasks are assigned to a partition.

Mapping tasks to processors and scheduling messages to wires is currently done separately. One could determine whether they can be combined to produce a mapping with fewer message cycles than if performed separately.

Finally, one can use the experience gained with mapping and scheduling to study the communication capabilities of various interconnection schemes for a set of hypothetical, but plausible, parallel computers. We started to do this in Chapter 3 but a comprehensive study is beyond the scope of this thesis. To pursue this one needs to develop a communication compiler that schedules messages to wires on more than just hypercubes.

BIBLIOGRAPHY

[1] F. Afrati, C. H. Papadimitriou, and G. Papadimitriou. The complexity of cubical graphs. *Information and Control*, pages 53–60, 1985.

[2] R. K. Agarwal and J. L. Richardson. Development of an Euler code on a Connection Machine. In H. D. Simon, editor, *Proc. of the Conference on Scientific Applications of the Connection Machine, NASA Ames Research Center, Moffett Field, California*, pages 27–63. World Scientific, September 12-14, 1988.

[3] F. André, J.-L. Pazat, and T. Priol. Experiments with mapping algorithms on a hypercube. In *Proc. of the* 4^{th} *Conference on Hypercubes, Concurrent Computers, and Applications*, pages 39–46, March 1989.

[4] T. J. Barth. On unstructured grids and solvers. In *Computational Fluid Dynamics, Lecture Series 1990-03.* Von Karman Instit., Belgium, March 1990.

[5] T. J. Barth and D. C. Jespersen. The design and application of upwind schemes on unstructured meshes. In *AIAA 1989,* 27^{th} *Aerospace Sciences Meeting*, January 1989. AIAA-89-0366.

[6] G. Bedrosian. private communication. 1986.

[7] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comp.*, 36(5):570–580, May 1987.

[8] F. Berman and L. Snyder. On mapping parallel algorithms to parallel architectures. *J. Parallel and Dist. Comput.*, 4:439–458, 1987.

[9] H. Berryman, J. Saltz, and W. Gropp. Krylov methods preconditioned with incompletely factored matrices on the CM-2. Technical Report TR-685,

Department of Computer Science, Yale University, New Haven CT, March 1989.

[10] S. N. Bhatt and I. C. F. Ipsen. How to embed trees in hypercubes. Technical Report YALEU/DCS/RR-443, Yale University, Yale University, New Haven CT, December 1985.

[11] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comp.*, 38(11):1526–1538, November 1989.

[12] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comp.*, 30(3):207–214, March 1981.

[13] S. H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. Soft. Eng.*, SE-7(6):583–589, Nov 1981.

[14] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* North-Holland, 1976.

[15] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, 1991.

[16] M. Y. Chan and F. Y. L. Chin. On embedding rectangular grids in hypercubes. *IEEE Trans. Comp.*, 37(10):1285–1288, October 1988.

[17] M. Chrobak and D. Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theoretical Computer Science*, pages 243–266, 1991.

[18] J. S. Clary, G. A. Howell, and Jr. S. L. Karman. Benchmark calculations with an unstructured grid solver on a SIMD computer. In *Proc. Supercomputing '89*, pages 32–41, Reno, NV, November 1983.

[19] E. D. Dahl. Mapping and compiled communication on the Connection Machine system. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Computer Conference*, Charleston, SC, April 1990. IEEE Computer Society Press, Los Alamitos, CA.

[20] E.D. Dahl. private communication, 1990.

[21] W. J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

[22] A. Edelman. Optimal matrix transposition and bit reversal on hypercubes: All-to-all personalized communication. Thinking Machines Corporation, 1990.

[23] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. In *Proceedings of the 3rd Hypercube Concurrent Computers and Applications Conference*, Pasadena, CA, January 1988.

[24] C. Farhat, N. Sobh, and K. C. Park. Dynamic finite element simulations on the Connection Machine. In Horst D. Simon, editor, *Proc. of the Conference on Scientific Applications of the Connection Machine, NASA Ames Research Center, Moffett Field, California*, pages 217–233. World Scientific, September 12-14, 1988.

[25] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. Technical Report 82CRD130, General Electric Co., Corporate Research and Development Center, Schenectady, NY, 1982.

[26] M. Fiedler. Algebraic connectivity of graphs. *Czech. Math. J.*, 23:298–305, 1973.

[27] M. Fiedler. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czech. Math. J.*, 25:619–633, 1975.

[28] Z. Galil, S. Micali, and H. Gabow. Priority queues with variable priority and an $o(ev \log v)$ algorithm for finding a maximal weighted matching in general graphs. In *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, pages 255–261, 1982.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[30] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Num. Anal.*, 10:345–363, 1973.

[31] E. N. Gilbert. Gray codes and paths on the n-cube. *The Bell System Technical Journal*, pages 815–826, May 1958.

[32] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *Int. J. Parallel Programming*, 16(6):427–449, 1987.

[33] S. K. Godunov. A finite difference method for the numerical computation of discontinuous solutions of the equations of fluid dynamics. *Matematicheskie Sbornik*, 47, 1959.

[34] S. W. Hammond and T. J. Barth. An efficient massively parallel Euler solver for 2-D unstructured grids. *AIAA*, April 1992.

[35] N. Hartsfield and G. Ringel. *Pearls in Graph Theory: a Comprehensive Approach.* Academic Press, 1990.

[36] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. National Bureau of Standards*, (49):409–436, 1952.

[37] D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985. Fourth Printing.

[38] C.-T. Ho and S. L. Johnsson. Embedding meshes in boolean cubes by graph decomposition. *J. Parallel and Dist. Comput.*, 8:325–339, 1990.

[39] K. E. Iverson. *A Programming Language.* Wiley, New York, 1962.

[40] D. C. Jespersen and C. Levit. Numerical simulation of flow past a tapered cylinder. In *AIAA 1991,* 29^{th} *Aerospace Sciences Meeting*, January 1991. AIAA-91-0751.

[41] S. L. Johnsson and C. T. Ho. Algorithms for matrix transposition on Boolean N-Cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9:419–454, 1988.

[42] S. L. Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comp.*, 38:1249–1268, 1989.

[43] B. W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–308, February 1970.

[44] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[45] D. W. Krumme, K. N. Venkataraman, and G. Cybenko. Hypercube embedding is NP-Complete. In M. Heath, editor, *First Hypercube Conference*, pages 148–157. SIAM, Knoxville, TN, August 1985.

[46] Ford L. R., Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[47] F. F. Lee. Partitioning of regular computation on multiprocessor systems. *J. Parallel and Dist. Comput.*, 9:312–317, 1990.

[48] S.-Y. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. Comp.*, C-36(4):433–442, April 1987.

[49] M. C. Levit. Grid communication on the Connection Machine: analysis, performance, and improvements. In Horst D. Simon, editor, *Proc. of the Conference on Scientific Applications of the Connection Machine, NASA Ames Research Center, Moffett Field, California*, pages 316–332. World Scientific, September 12-14, 1988.

[50] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comp.*, 37(11):1384–1397, November 1988.

[51] L. N. Long. A three-dimensional Navier-Stokes method for the Connection Machine. In H. D. Simon, editor, *Proc. of the Conference on Scientific Applications of the Connection Machine, NASA Ames Research Center, Moffett Field, California*, pages 64–93. World Scientific, September 12-14, 1988.

[52] G. A. Mago. A network of computers to execute reduction languages. *Int. J. Comput. Inform. Sci.*, 1979.

[53] O. A. McBryan. Connection Machine application performance. In Horst D. Simon, editor, *Proc. of the Conference on Scientific Applications of the Connection Machine, NASA Ames Research Center, Moffett Field, California*, pages 94–115. World Scientific, September 12-14, 1988.

[54] J. Myczkowski, M. Bromley, and D. McGowan. Extremely fast difference techniques for the Connection Machine. In *AIAA 1991*, 29th *Aerospace Sciences Meeting*, January 1991. AIAA-91-0436.

[55] A. Pothen, H. D. Simon, and K-P Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, July 1990.

[56] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. Comp.*, 36(7):845–858, July 1987.

[57] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *J. Comput. Phys.*, 43:357–372, 1981.

[58] Y. Saad and M. H. Schultz. Topological properties of hypercubes. Technical Report YALEU/DCS/RR-389, Yale University, New Haven, CT, 1985.

[59] Y. Saad and M. H. Schultz. Data communication in hypercubes. *J. Parallel and Dist. Comput.*, 6:115–135, 1989.

[60] P. Sadayappan, F. Ercal, and J. Ramanujam. Cluster partitioning approaches to mapping parallel programs onto a hypercube. Technical report, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1988. Submitted to Parallel Computing.

[61] P. Sadayappan, F. Ercal, and J. Ramanujam. Parallel graph partitioning on a hypercube. Technical Report OSU-CISRC-3/88-TR8, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1988.

[62] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multiprocessors. Technical Report Report 91-12, ICASE, 1991.

[63] J. Savage and M. Wloka. Heuristics for parallel graph-partitioning. Technical Report CS-89-41, Dept. of Computer Science, Brown University, Providence,

RI, January 1991.

[64] J. Savage and M. Wloka. Parallel graph-embedding and the Mob heuristic. Technical Report CS-91-07, Dept. of Computer Science, Brown University, Providence, RI, February 1991.

[65] R. Schreiber. An assessment of the Connection Machine. Technical Report TR90.40, RIACS, NASA Ames Research Center, Moffett Field, CA, June 1990.

[66] K. Schwan, W. Bo, N. Bauman, P. Sadayappan, and F. Ercal. Mapping parallel applications to a hypercube. In M. Heath, editor, *Hypercube Multiprocessors 1987*, pages 141–151, Knoxville, TN, Sept 1987. SIAM.

[67] J. T. Schwartz. Ultracomputers. *ACM Trans. Programming Language Syst.*, 2(4):484–521, October 1980.

[68] C.-C. Shen and W.-H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Trans. Comp.*, 34(3):197–203, March 1985.

[69] M. K. Shephard and M. K. Georges. Automatic thre-dimensional mesh generation by the finite octree technique. *Int. J. Numer. Meth. Engng.*, 32(4):709–749, 1991.

[70] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[71] J. B. Sinclair. Efficient computation of optimal assignments for distributed tasks. *J. Parallel and Dist. Comput.*, 4(4):342–362, August 1987.

[72] J. S. Squire and S. M. Palais. Programming and design considerations for a highly parallel computer. In *AFIPS Cong. Proc.*, number 23, pages 395–400,

1963.

[73] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Soft. Eng.*, SE-3(1):85–93, Jan 1977.

[74] R. C. Strawn. Wing tip vortex calculations with an unstructured adaptive-grid Euler solver. In *Proceedings of the* 47th *Annual Forum of the American Helicopter Society*, Phoenix, AZ, May 1991.

[75] E.-G. Talbi and P. Bessière. A parallel genetic algorithm for the graph partitioning problem. In *Proc. of the International Conference on Supercomputing*, Cologne, June 1991.

[76] Thinking Machines Corporation. *CMSSL Release Notes*, June 1991. Version 2.2.

[77] C. Tong. The preconditioned conjugate gradient on the Connection Machine. In Horst D. Simon, editor, *Proc. of the Conference on Scientific Applications of the Connection Machine, NASA Ames Research Center, Moffett Field, California*, pages 188–213. World Scientific, September 12-14, 1988.

[78] L. G. Valiant. A scheme for fast parallel communication. *SIAM J. Computing*, 11(2):350–361, May 1982.

[79] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc.* 13th *ACM Symposium of Theory of Computing*, pages 263–277, 1981.

[80] V. Venkatakrishnan, H. D. Simon, and T. J. Barth. A MIMD implementation of a parallel Euler solver for unstructured grids. Technical Report RNR-91-024, Applied Research Branch, NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, September 1991.

[81] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Technical Report C3P913, Concurrent Supercomputing Facility, California Institute of Technology, Pasadena, CA, June 1990.

[82] S. Yalamanchili and D. T. Lee. A mapping algorithm for multiprocessor architectures. In *Proceedings of the 26th Allerton Conference on Communications, Control and Computing*, Honeywell Systems Research Center, Minneapolis, MN, September 1988.

APPENDIX A
TEST GRIDS

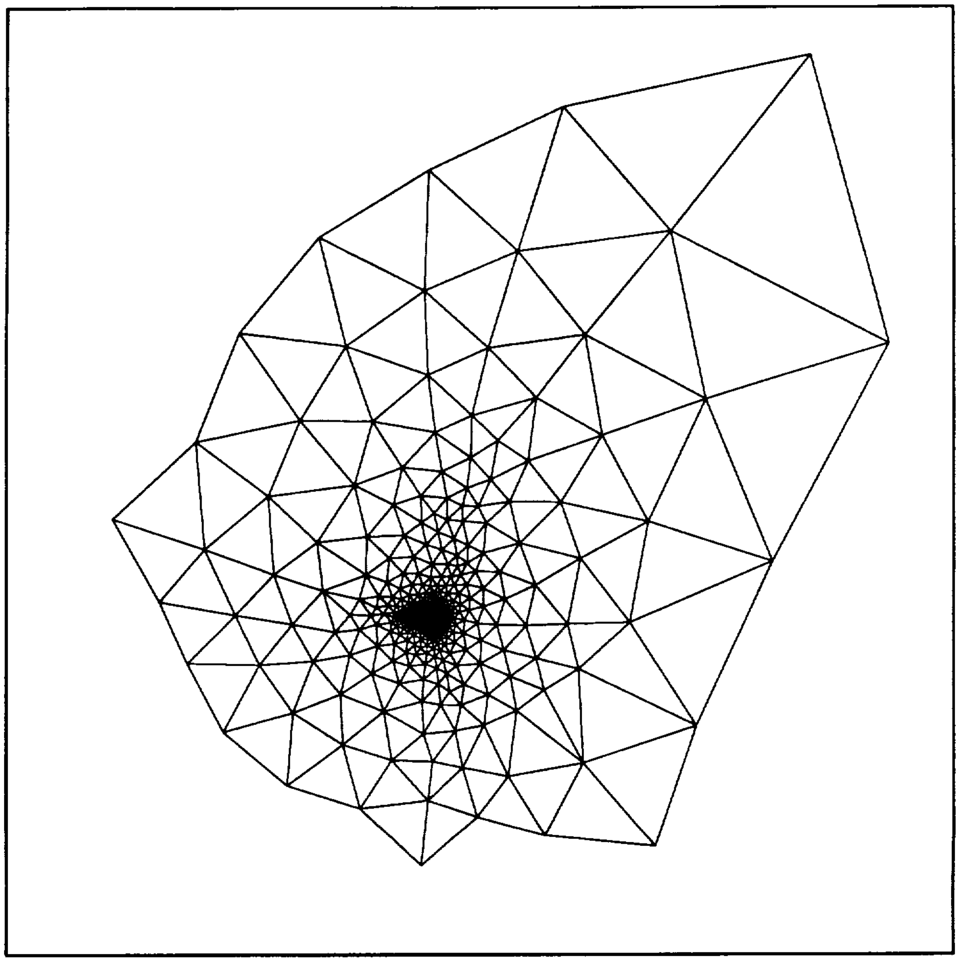


Figure A.1: Test case 3elt: Unstructured mesh about 3 component airfoil with flaps down.

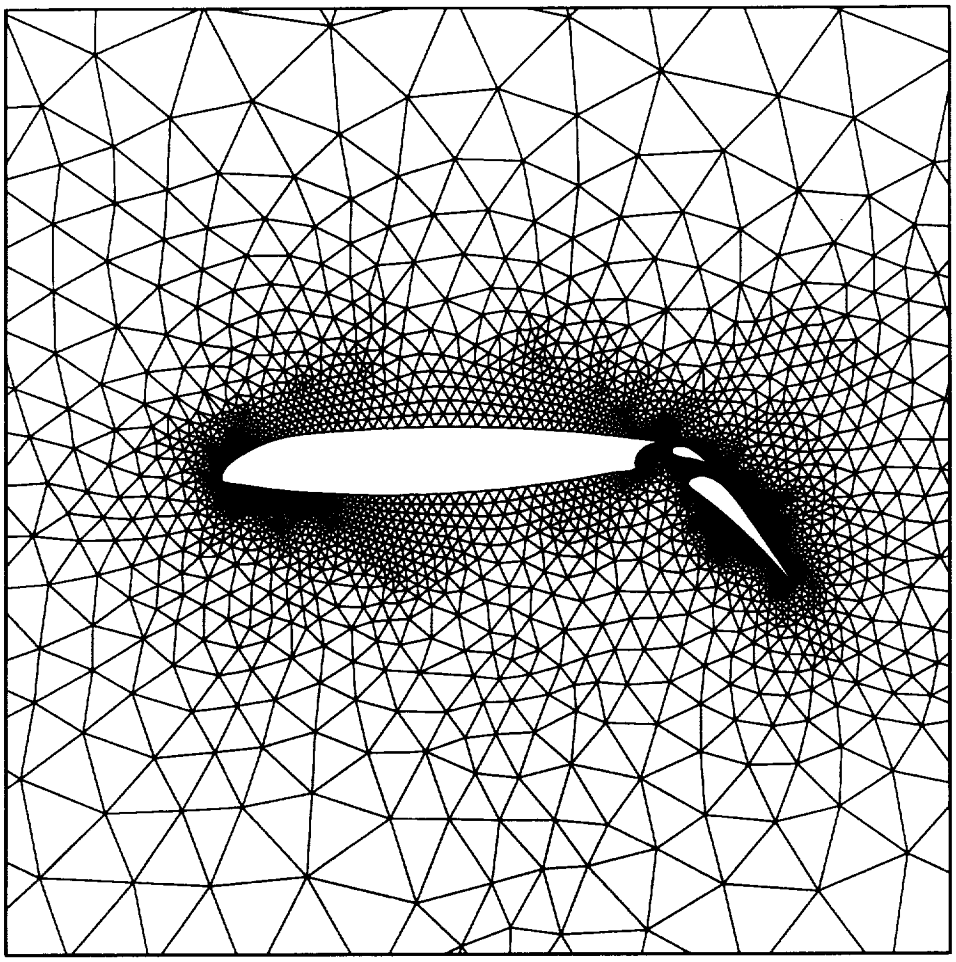


Figure A.2: **Test case 3elt: Closeup of unstructured mesh about 3 component airfoil with flaps down.**

degree	num. vertices
3	6
4	425
5	258
6	3791
7	232
8	6
9	2

Table A.1: Histogram of vertex degrees for 3elt.

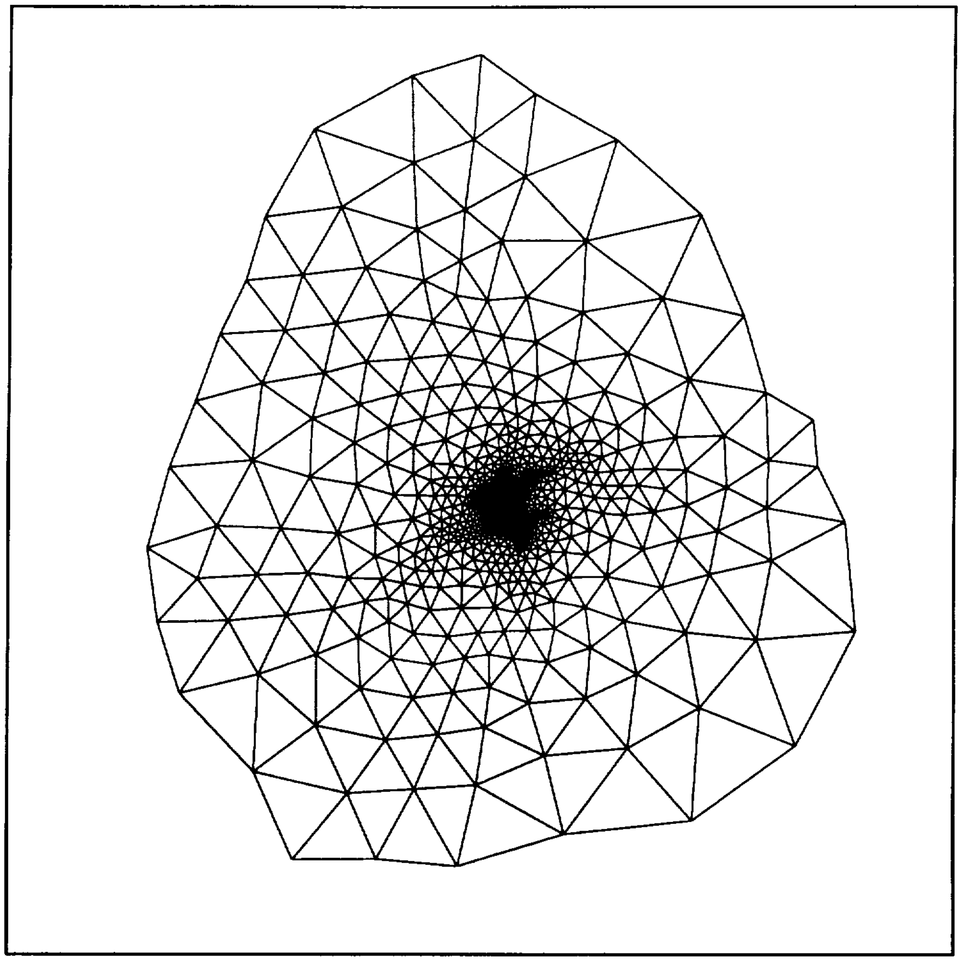


Figure A.3: Test case 4elt: Mesh about 4 component airfoil with extended flaps.

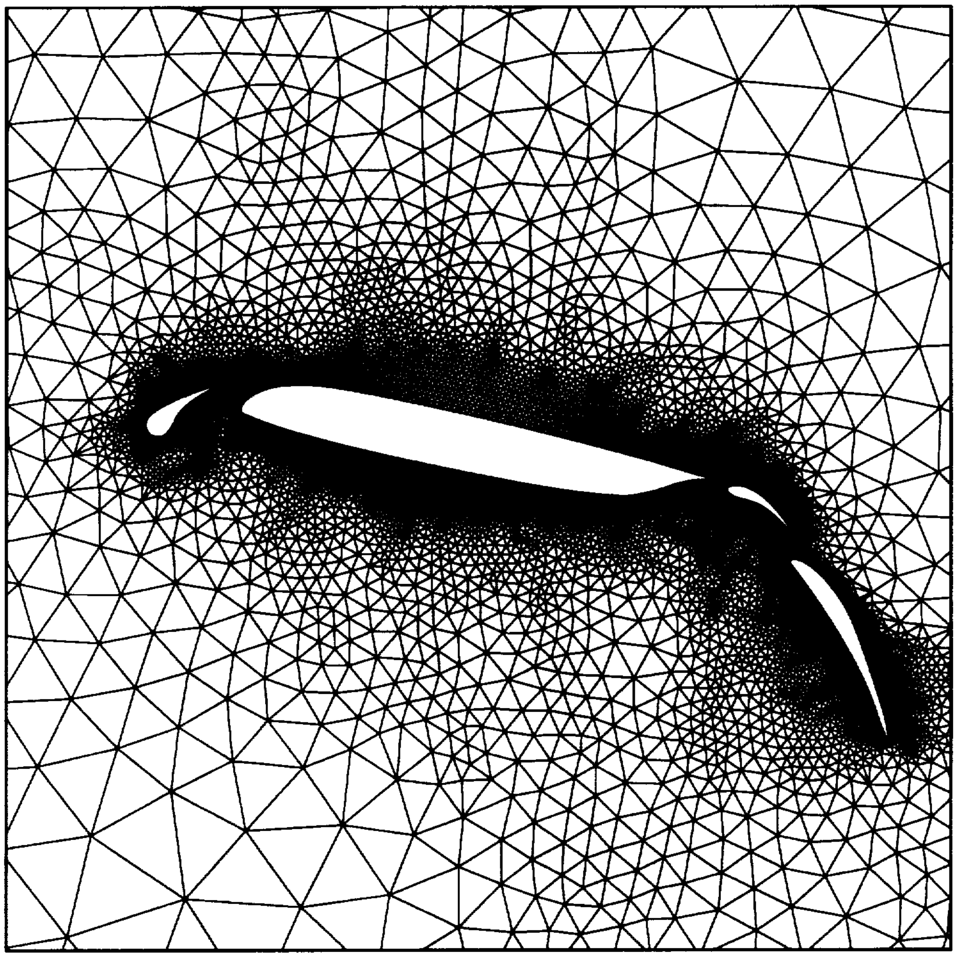


Figure A.4: **Test case 4elt: Closeup of mesh about 4 component airfoil with extended flaps.**

degree	num. vertices
3	4
4	934
5	755
6	13189
7	699
8	20
9	4
10	1

Table A.2: Histogram of vertex degrees for 4elt.

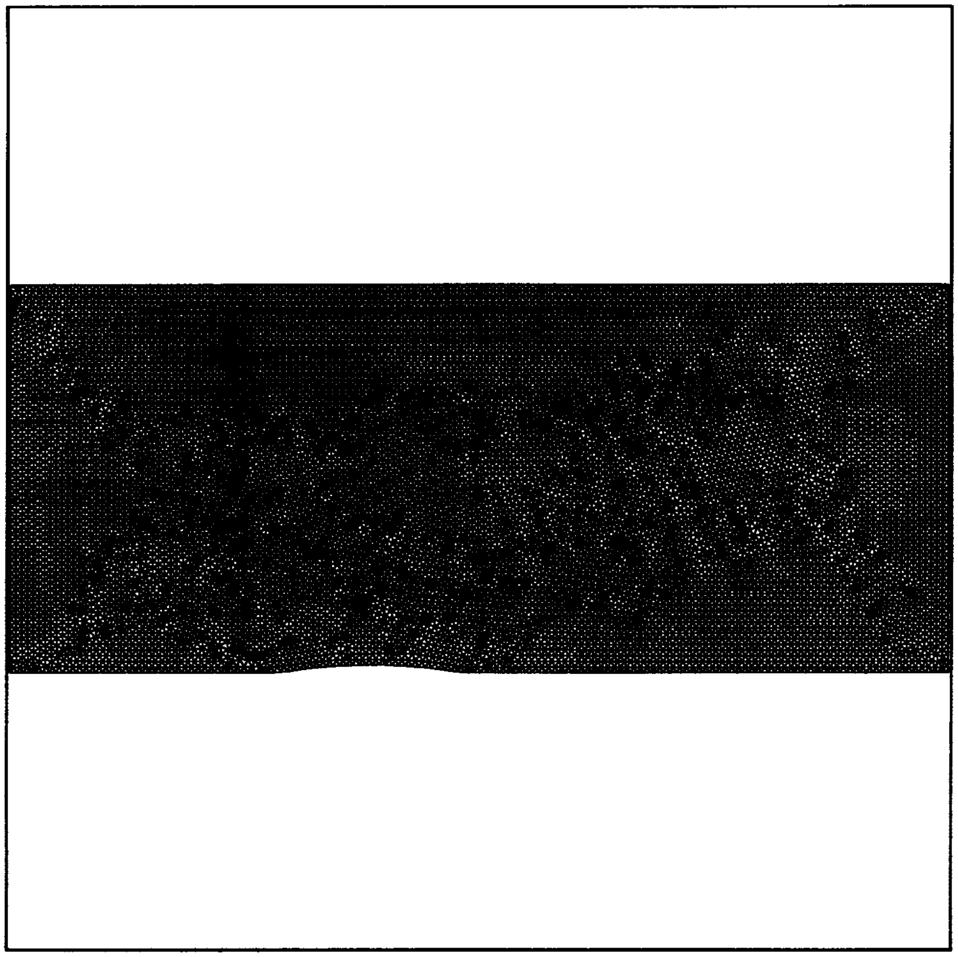


Figure A.5: Test case bump: Unstructured mesh over a bump.

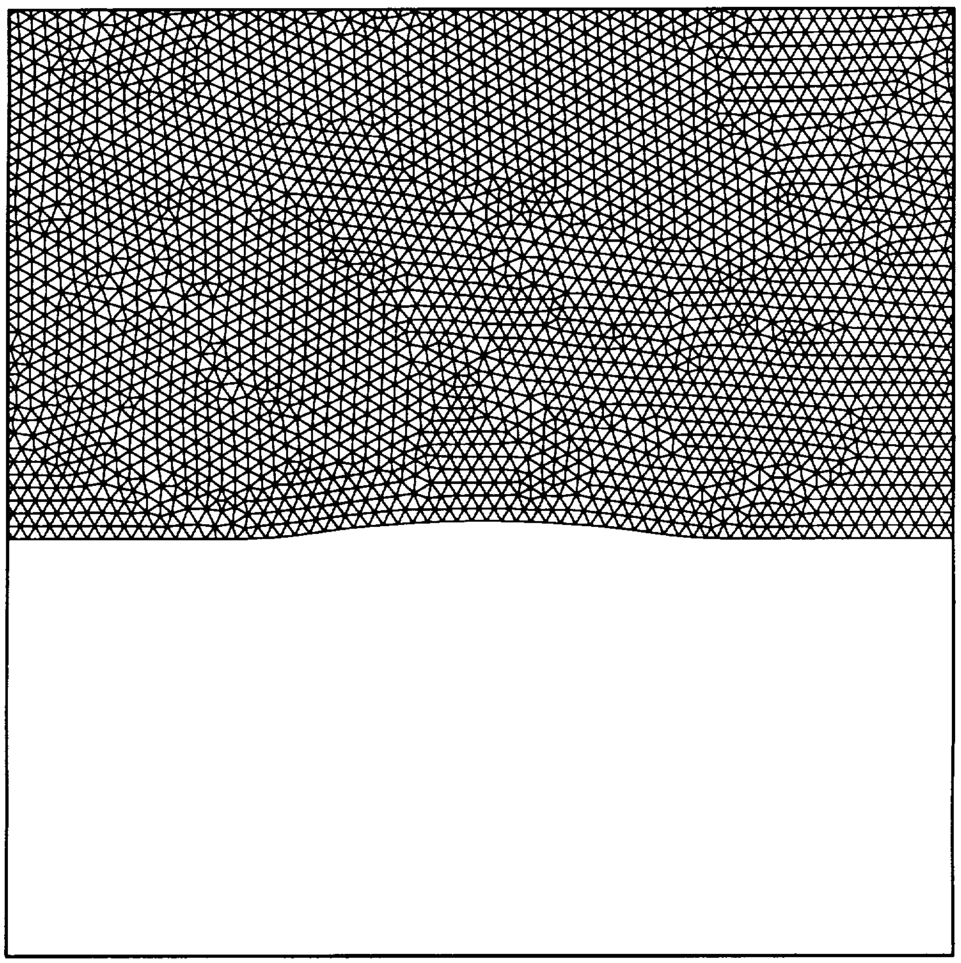


Figure A.6: Test case bump: Closeup of unstructured mesh over a bump.

degree	num. vertices
3	10
4	399
5	498
6	8396
7	490
8	7

Table A.3: Histogram of vertex degrees for bump.

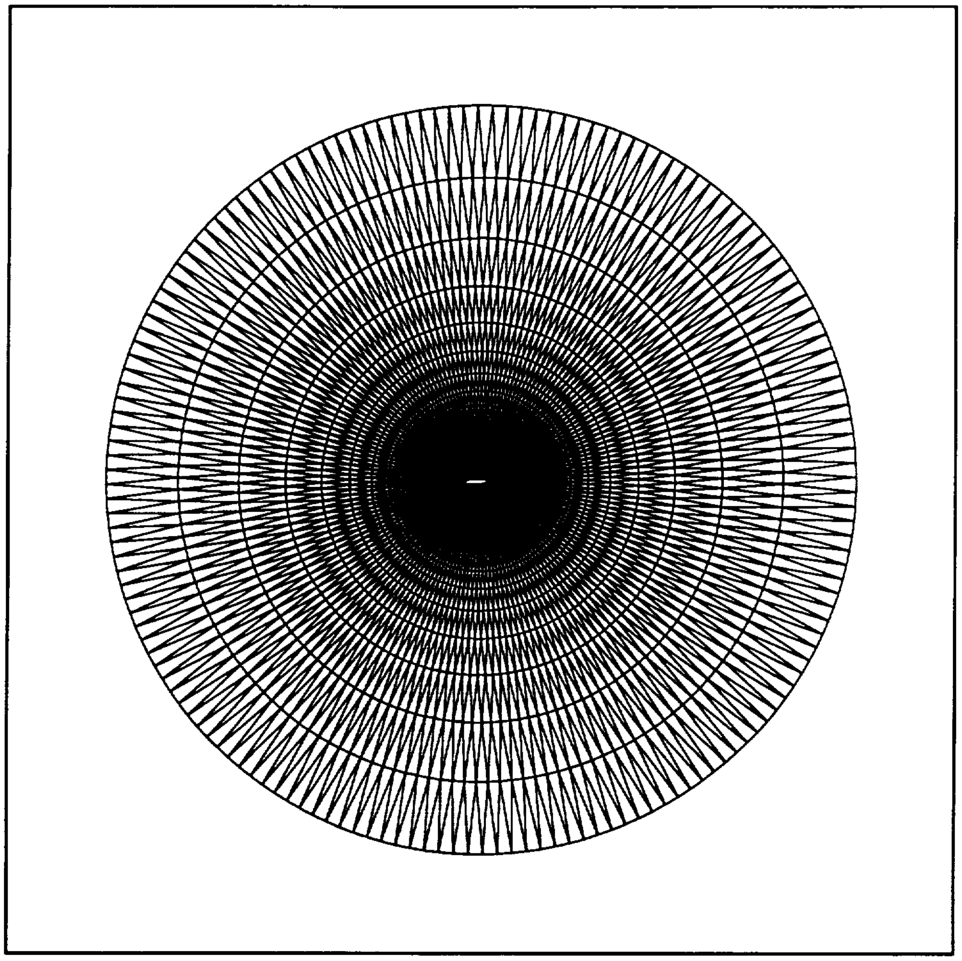


Figure A.7: Test case 4elt-2: Unstructured mesh about 4 component airfoil with flaps down.

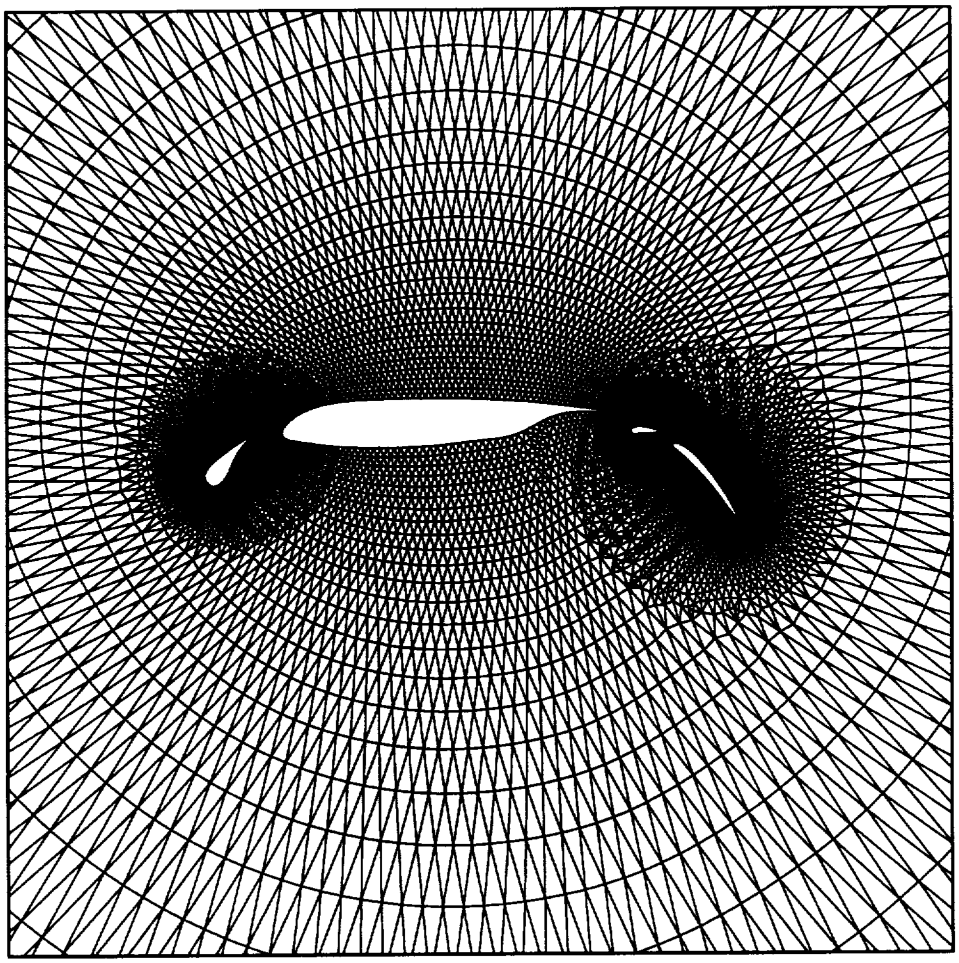


Figure A.8: Test case 4elt-2: Closeup of unstructured mesh about 4 component airfoil with flaps down.

degree	num. vertices
3	35
4	952
5	1334
6	7230
7	1195
8	292
9	82
10	21
12	2

Table A.4: Histogram of vertex degrees for 4elt-2.

degree	num. vertices
7	4
9	498
12	118
15	780
17	1648
18	2
19	2374
22	1
24	2
25	59
26	12
27	2
28	101
30	105
31	1
32	412
34	107
36	210
38	29
40	40
42	5
44	7

Table A.5: **Histogram of vertex degrees for Permanent Magnet Motor, test case Motor.**

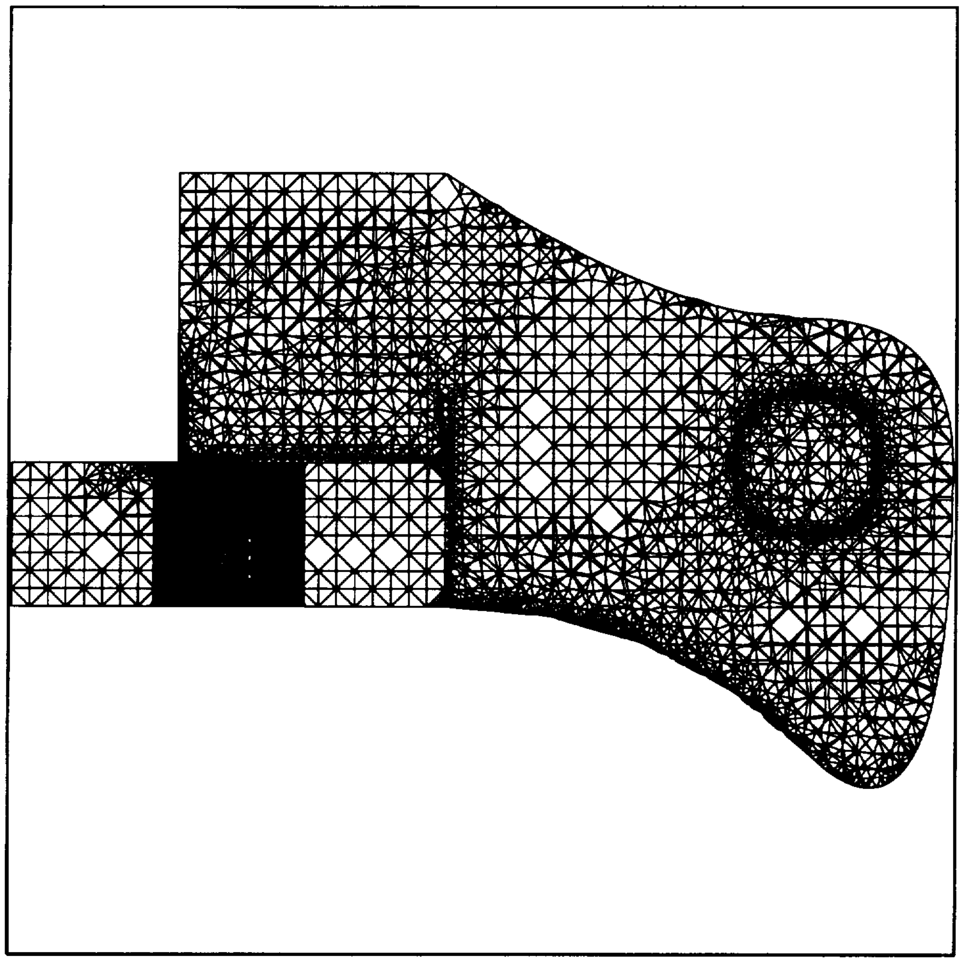


Figure A.9: Test case bracket: Tetrahedral mesh of a bracket, view a., only surface elements shown.

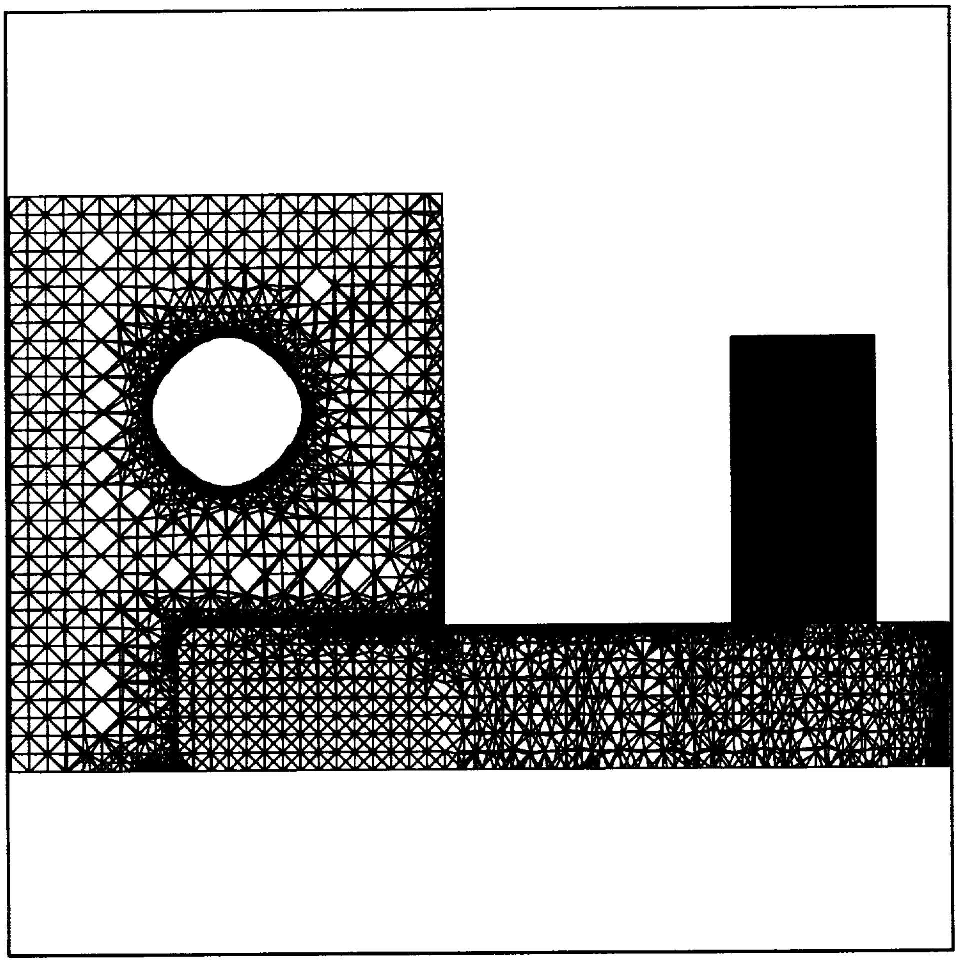


Figure A.10: Test case bracket: Tetrahedral mesh of a bracket, view b., only surface elements shown.

degree	num. vertices
3	10
4	250
5	3831
6	10285
7	2696
8	3535
9	3396
10	3525
11	4321
12	3121
13	6217
14	3189
15	2413
16	2521
17	1982
18	5472
19	2060
20	2019
21	855
22	348
23	319
24	137
25	60
26	41
27	14
28	7
29	2
30	2
31	1
32	1

Table A.6: Histogram of vertex degrees for bracket.

degree	num. vertices	degree	num. vertices
5	1	36	43
6	56	37	36
7	301	38	26
8	1598	39	16
9	4594	40	24
10	10083	41	19
11	9212	42	18
12	8274	43	10
13	6521	44	6
14	16523	45	8
15	4170	46	12
16	2616	47	1
17	1967	48	7
18	1363	49	5
19	1156	50	6
20	833	51	3
21	627	52	4
22	486	53	5
23	400	54	3
24	322	55	1
25	265	56	3
26	213	57	2
27	213	59	1
28	152	61	2
29	113	64	1
30	115	68	1
31	85	74	1
32	66	77	1
33	56	78	1
34	53	125	1
35	42		

Table A.7: Histogram of vertex degrees for helicopter blade, test case rotor.

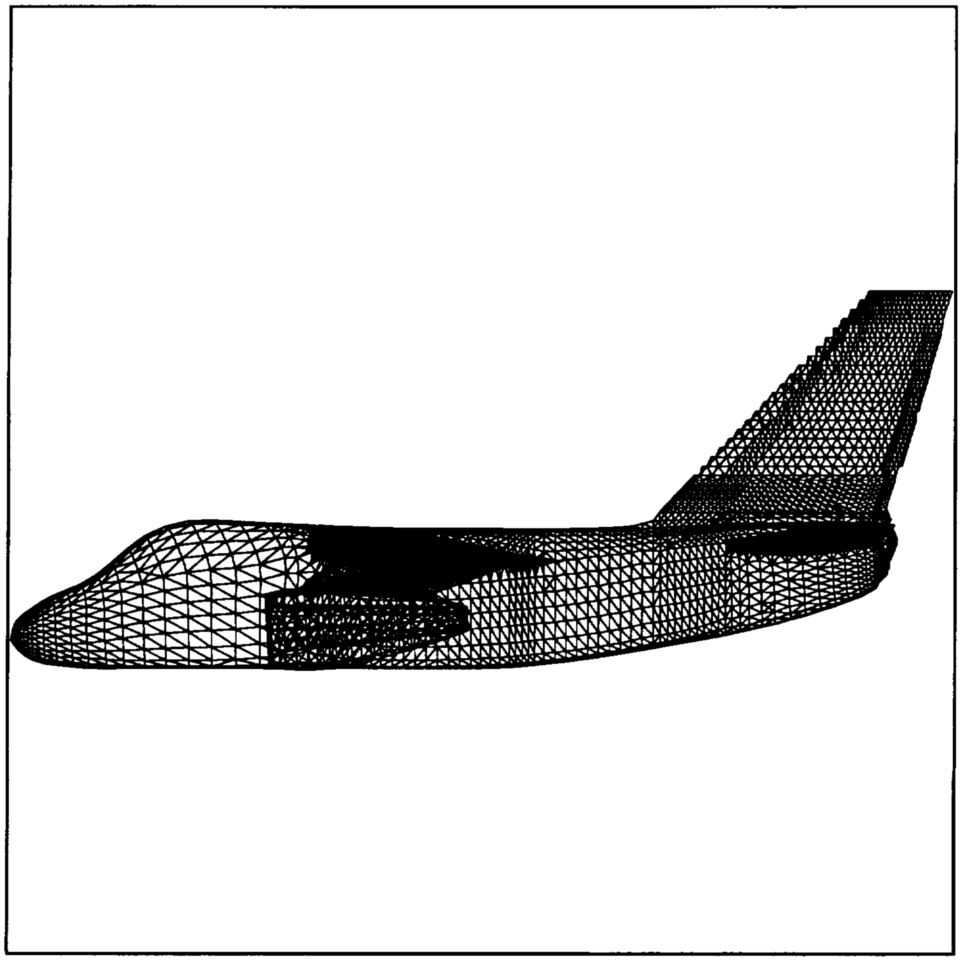


Figure A.11: Test case viking: closeup of 3D unstructured tetrahedral mesh about Lockheed S-3A Viking aircraft, only surface elements shown.

degree	num. vertices	degree	num. vertices
3	6	24	98
4	56	25	92
5	105	26	66
6	275	27	43
7	1388	28	46
8	3027	29	42
9	5856	30	22
10	5071	31	16
11	47379	32	22
12	7116	33	11
13	3821	34	6
14	38726	35	4
15	3789	36	3
16	2911	37	5
17	19943	38	3
18	1589	39	2
19	1288	40	1
20	12247	41	1
21	760	42	1
22	206	43	1
23	272	44	1

Table A.8: Histogram of vertex degrees for 3D unstructured tetrahedral mesh about Lockheed S-3A Viking aircraft.